AFRL-IF-RS-TR-2006-237
Final Technical Report
July 2006

INCREASING INTRUSION TOLERANCE VIA SCALABLE REDUNDANCY

Carnegie-Mellon University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. S469

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* JULY 2006	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Jun 04 – Dec 05

4. TITLE AND SUBTITLE INCREASING INTRUSION TOLERANCE VIA SCALABLE REDUNDANCY	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER FA8750-04-1-0238
	5c. PROGRAM ELEMENT NUMBER 62301E
6. AUTHOR(S) Michael K. Reiter and Gregory R. Ganger	**5d. PROJECT NUMBER** S469
	5e. TASK NUMBER SR
	5f. WORK UNIT NUMBER SP
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Carnegie-Mellon University, 5000 Forbes Ave., Pittsburgh PA 15213	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) DARPA, 3701 N. Fairfax Dr, Arlington VA 22203-1714; AFRL/IFGB, 525 Brooks Rd, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2006-237

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA # 06-496

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This final technical report presents a novel protocol to achieve invariability of nested method invocations. This protocol tolerates Byzantine faulty clients, as is necessary when methods are invoked from other distributed objects that themselves must withstand Byzantine faults. This final report further presents a detail of this protocol, an argument about its correctness and a description of its implementation and performance in a distributed object system.

15. SUBJECT TERMS
Protocol, PASIS, Byzantine, fault-scalability

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Robert Vaeth
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UL	83	**19b. TELEPONE NUMBER (*Include area code)***

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This report summarizes the results of the work on the DARPA under project entitled INCREASING INTRUSION TOLERANCE VIA SCALABLE REDUNDANCY Air Force contract FA8750-04-1-0238. The project worked toward dramatically increasing the scalability of fault- and intrusion-tolerant services, particularly in the efficiency of tolerating significant numbers of failures and compromises. This project has built upon foundational work laid in two previous, independent and concurrent DARPA projects: Information Processing Technology Office (IPTO) PASIS project for implementing survivable data storage, and Advance Technology Office (ATO) Fleet project for implementing an intrusion-tolerant object-oriented system, integrating the lessons and advances of these individual projects as a basis for further innovation. Indeed, it is from this cross-pollination that many of the advances have derived. Specifically, we have pushed the state-of-the-art in intrusion-tolerant distributed services in following ways.

1. We developed novel protocols for distributed data storage that offer linearizable data access, wait-free liveness, tolerance of Byzantine client and server failures (corruptions), and far superior latency and throughput as the system scales. These protocols exploit server versioning, quorums, and lazy verification of updates to achieve their performance, especially in the common cases of no concurrency or attacks. Relative to the state-of-the-art, our protocols provide at least a threefold improvement in access latency, even for relatively small numbers of servers, a gap that grows as the number of servers increases. Simultaneously, they increase by a factor of five the throughput achieved by the previous state-of-the-art, even for relatively small server configurations, and also scale better as the system grows.

2. We developed protocols to support fault-scalable linearizable access to services. For example, directories are services that have a specific structure and a small set of legal operations (e.g., inserting a new directory entry rather than opaque writes). Because of the additional semantic information available about what constitutes a valid operation (arbitrary overwriting is disallowed), our protocols can constrain even corrupt clients form destroying the integrity and consistency of directory structures. We observe throughput improvements of a factor of four when tolerating five Byzantine faults, compared to the previous state-of-the-art.

3. We developed protocols for highly resilient distributed object-based systems that offer linearizable method invocations, including nested invocations in which one object is invoked by another, where some replicas of each may be corrupt. The result is unprecedented scalability for general intrusion-tolerant service construction. Our protocols provide sublinear growth in access cost as the number of object replicas grows, as compared to the linear-or-worse access costs offered by existing approaches.

1.1 Approach / Overview of the Problem

This project targeted dramatic increases in the scalability of intrusion-tolerant services.

At the core of such services are the protocols used to invoke server functions, and our goal was to develop new protocols that scale better than the state-of-the-art. Doing so enables creation of intrusion-tolerant services that can survive large numbers of server corruptions.

Distributed services typically utilize redundant state across servers to tolerate faults. When the fault model includes arbitrary corruption (Byzantine failure) of individual servers, the predominant form of distributed service implementation is *state machine replication* [Schneider90]. In this approach, all servers retain identical state and process all client requests in the same order, thereby keeping all server states synchronized. However, this approach scales poorly: First, since it does not permit load dispersion among servers, adding additional servers cannot enable further throughput (and hence scaling in the number of client requests); instead, it exacerbates the cost of ensuring the same processing order across all servers. Second, state machine replication offers little opportunity to move processing load from servers to clients. Typically client processing is limited to simply obtaining return results from individual servers and accepting the response received from a majority of servers.

The research program we developed to address these issues consisted of three progressive tasks, each of which leveraged and built upon the ones before it, with the goal of efficiently integrating the techniques of versioning and Byzantine quorums into fault- and intrusion-tolerant protocols for efficiently and scalably supporting increasingly general service types. The three tasks were:

- To develop, prove, implement and experiment with protocols that scale to highly fault- and intrusion-tolerant access to read-write data storage with much better efficiency than today's state-of-the-art.

- To develop, prove, implement and experiment with protocols that scale to highly fault- and intrusion-tolerant access to structured data objects (e.g., directories and indices) with much better efficiency than today's state-of-the-art.

- To develop, prove, implement and experiment with protocols that scale to highly fault- and intrusion-tolerant access to arbitrary deterministic objects (as in object-based programming systems), including support for nesting of object calls, with much better efficiency than today's state-of-the-art.

1.2 Contributions

Papers documenting this research have been accepted at recognized conferences, including the International Conference on Distributed Computing Systems [Goodson04a], the IEEE Symposium on Reliable Distributed Systems [Fry04, Abd-El-Malek05a], and the ACM Symposium on Operating Systems Principles [Abd-El-Malek05c]. Additional publications are in preparation. Carnegie Mellon Technical Reports (e.g., [Abd-El-Malek05b]) and a Doctoral thesis [Wylie05] have also been published with additional details.

1.2.1 Read-write Storage

Building on the PASIS architecture for survivable storage, one of the first tasks addressed in this research was to develop a protocol that uses versioning to implement linearizable [Herlihy90] read-write storage despite the arbitrary corruption of a limited number of servers and even clients with authority to perform reads and writes on the data. (Versioning is additionally useful for recovering from written values that are incorrect in the context of the application [Strunk00], as may be necessary when corrupted clients overwrite data.) Our protocol provides a very strong liveness property called *wait freedom* [Herlihy91], which intuitively means that each client can drive its own reads and writes to completion in finitely many steps and without help from other clients. The protocols provided unprecedented efficiency and scalability for fault-tolerant services, via the combination of versioning and Byzantine quorums.

We began our development of a Byzantine fault-tolerant read/write protocol by describing a decentralized consistency protocol for survivable storage that exploits local data versioning within each storage-node.

Such versioning enables the protocol to efficiently provide linearizability and wait-freedom of read and write operations to erasure-coded data in asynchronous environments with Byzantine failures of clients and servers. By exploiting versioning storage-nodes, the protocol shifts most work to clients and allows highly optimistic operation: reads occur in a single round-trip unless clients observe concurrency or write failures. Measurements of a storage system prototype using this protocol show that it scales well with the number of failures tolerated, and its performance compares favorably with an efficient implementation of Byzantine-tolerant state machine replication.

To facilitate the construction of a versatile storage infrastructure, a set of related protocols has been developed for reading and writing data objects called the Read/Write Protocol Family (R/W-PF). The R/W-PF provides versatility: objects with different per-object resiliency requirements can be stored in the same storage infrastructure. The costs (response time, number of servers required, etc.) of storing an object are commensurate with its resiliency requirements. The R/W-PF incorporates versatile storage mechanisms, such as erasure codes, witnesses, and quorums, in its design, allowing the efficiency of read and write access to stored objects to be tuned to meet capacity and performance requirements. Measurements of PASIS, a prototype storage system based on the R/W-PF, demonstrate its versatility as well as fault-scalability.

Verification of write operations is a crucial component of Byzantine fault-tolerant consistency protocols for storage. Lazy verification shifts this work out of the critical path of client operations. This shift enables the system to amortize verification effort over multiple operations, performing verification during otherwise idle time, with only a sub-set of storage-nodes carrying out the task. Measurements of lazy verification in a Byzantine fault-tolerant distributed storage system show that the cost of verification can be hidden completely from both the client read and write operation in workloads with idle periods. Furthermore, in workloads without idle periods, lazy verification amortizes the cost of verification over many versions and so provides a factor of four higher write bandwidth when compared to performing verification during each write operation.

1.2.2 Richer Specialized Objects – the Query/Update Protocol

We have developed protocols for selected services (objects) with richer types of operations than simple reads and writes, such as insertions into a directory or allocations of a resource. We have extended the scalable and efficient read-write protocols to support linearizable access to this type of object. For example, directories are objects of a specific structure that are accessed using operations other than opaque reads and writes (e.g., inserting a new directory entry). Because of the additional semantic information available about what constitutes a valid operation on a directory (e.g., arbitrary overwriting is disallowed), correct servers will use this information to constrain even corrupt clients from destroying the consistency of directory structures.

A fault-scalable service can be configured to tolerate increasing numbers of faults without significant decreases in performance. Our Query/Update (Q/U) protocol enables the construction of fault-scalable Byzantine fault-tolerant services. The optimistic quorum-based nature of the Q/U protocol allows it to provide better throughput and fault-scalability than replicated state machines using agreement-based protocols. A prototype service built using the Q/U protocol outperforms the same service built using a popular replicated state machine implementation at all system sizes in experiments that permit an optimistic execution. Moreover, the performance of the Q/U protocol decreases by only 36% as the number of Byzantine faults tolerated increases from one to five, whereas the performance of the replicated state machine decreases by 83%.

1.2.3 Nested Objects

We have developed protocols to implement linearizable access to arbitrary deterministic objects (i.e., objects implementing any function), both from centralized clients and other distributed objects of which a limited number of replicas may be corrupt. These protocols have been developed for object-oriented systems in which one distributed object may be "passed into" another as an argument in a method invocation. We have developed a framework that supports nested method invocations among Byzantine fault-tolerant, replicated objects that are accessed via quorum systems. A challenge in this context is that client object replicas can induce unwanted method invocations on server object replicas, due either to redundant invocations by client replicas or Byzantine failures within the client replicas. At the core of our framework are a new quorum-based authorization technique and a novel method invocation protocol that ensure the linearizability of nested method invocations despite Byzantine client and server replica failures.

1.3 Technology Transfer

1.3.1 Impact

Our protocols provide significant efficiency and scalability benefits over today's state-of-the-art approaches to fault- and intrusion-tolerance. For example, we have shown threefold latency improvements for relatively small configurations (e.g., those tolerating 3–5 Byzantine server failures), with improvements increasing as system sizes and fault-tolerance grow. Likewise, a twofold improvement in throughput has been seen, again growing with system size because of superior scalability. Scalability also shows up in terms of efficiency retention as the system scales in all major dimensions: number of servers, number of failures tolerated, and number of clients.

Such large improvements are critical to making highly fault-tolerant and intrusion-tolerant services practical; without them, such tolerances are limited to small-scale systems providing only simple types of services while still incurring substantial performance costs.

1.3.2 Technology Transfer Avenues

A critical aspect of all DoD-funded work is technology transfer to military customers. For infrastructure research, direct transfer is extremely difficult, because it is inappropriate to experiment with critical military infrastructure in real environments—it is unthinkable to endanger military operation with infrastructure technology that has not been through rigorous, production-quality testing. This creates a dangerous tension, since the best role for DoD-funded research is exploration of risky, long-horizon new concepts—approaches to building infrastructures, like those described herein. Over the years, CMU's infrastructure researchers have developed a proven approach to technology transfer, based on a combination of open source software and industry consortia. Of course, sharing software prototypes publicly allows others to replicate our experiments and build on the work. To ensure that our research results transfer to military systems, however, we work closely with the industry that provides military infrastructure. We have a close relationship and frequent interactions with industry leaders through our solid consortia in storage systems and security. Such interactions achieve the kind of technology transfer truly needed for new infrastructure approaches to become reality: integration into COTS components.

Along with publishing our results in several recognized, juried conferences, over the course of this research we have spent large amounts of time explaining our fault- and intrusion-tolerant services to industry leaders from the many companies who support our research (including EMC Corporation, EqualLogic, Inc., Hewlett-Packard Labs, Hitachi, IBM, Intel Corporation, Microsoft Research, Network Appliance, Oracle Corporation, Panasas, Inc., Seagate Technology, Sun Microsystems and Veritas). These interactions have refined the practicality of our work and its acceptance by the various companies. Important meetings with our sponsors include the annual Parallel Data Lab (http://www.pdl.cmu.edu) Retreat and Open House events and the Cylab (http://www.cylab.cmu.edu) courses and Open House events, during which technical leaders from member companies hear about and discuss our research activities. Based on these interactions, we expect to see several of the concepts explored in this research appearing in products available to DoD users in the near term and we feel it is the most effective way for technology transfer of infrastructure research projects to achieve DoD deployment.

1.4 Red Teaming of R/W Storage Protocol

On December 2, 2005, a red team exercise was held at Carnegie Mellon to assess the R/W storage protocol developed as part of this project. The exercise was designed and run by a group from Sandia National Laboratories, including Kandy Phan and Michael Collins. It was also attended by five CMU researchers: PI Mike Reiter, Co-PI Greg Ganger, Michael Abd-El-Malek, Pratyusa Manadhata and Oren Dobzinski. Over the course of the day, possible attacks on the protocol and systems using the protocol were explored. The exercise increased our confidence in the proven correctness of the protocols and their value, as no new attacks (beyond those laid out as assumptions in the papers) were identified. The final report on the red teaming exercise is not yet available.

1.5 Document Overview

The rest of this document is structured as follows: Chapter 2 presents a high-level overview of the technology. Chapter 3 reviews the read/write protocol. Chapter 4 describes the query/update protocol and Chapter 5 explores support for nested objects.

2 TECHNOLOGY OVERVIEW

2.1 Background to Our Technical Approach

In this section we describe the techniques from which we build, and the manner in which we combined and advanced them to achieve unprecedented scalability in intrusion-tolerant services. Section 2.2.1 begins by describing various challenges we faced and the techniques from which we innovated during our research. Background on the PASIS and Fleet systems which were the building blocks of this project is provided in Section 2.1.2.

2.1.1 Techniques

Distributed services, linearizability, and wait freedom. Distributed services employ multiple servers in order to survive failures and/or corruptions. State for the service is spread redundantly across servers, and a client interacts with some number of servers in order to obtain service. When server corruptions are to be tolerated, it is generally necessary for the client to interact with multiple servers and to use the responses from multiple servers to corroborate a response[1].

A central challenge in implementing distributed services is ensuring proper service semantics in the face of concurrent accesses and client failures (even benign ones). Concurrent accesses to the service can drive server states apart, due to servers processing non-commutative operations in different orders. Client failures can result in some operations being performed only partially, i.e., at fewer than the required number of servers. Ideally, and despite these problems, the service should *appear* to the client application as a centralized one in which each operation occurs in isolation and completely. The strongest and most commonly accepted formalization of this property for a service (or object) is *linearizability* [Herlihy90]. Informally, linearizability requires that the return results of all operations are consistent with some sequential execution of these operations, in which each is performed at a distinct point in real time between the moment the client began the operation and the time the client received its response.

Outside our work, all systems of which we are aware that implement linearizable services in the face of Byzantine server failures do so by imposing the linearization order explicitly at the time the operation is submitted, and then enforcing that all (correct) servers process requests in this linearization order. In this way, all correct servers execute in lock-step and retain identical states (presuming the service is deterministic), and therefore remain capable of corroborating one another's responses. This approach is often called *state machine replication* [Schneider90], and the mechanism that imposes the linearization order on client requests is often called *atomic broadcast*.

The strongest generally considered liveness condition for linearizable implementations is *wait freedom* [Herlihy91, Jayanti98]. Informally, in our model, wait freedom requires that each correct client can drive its operation to completion in finitely many steps, assuming only some limit on the number of servers that fail (and that correct servers make progress). In particular, operation completion does not require that *other* clients make progress; as such, it intuitively precludes the use of locking to enforce linearizability, if one client must drop locks for another to perform its operation. We emphasize that wait-freedom is a very strong property that ensures progress absent any timing assumptions about the rate at which processes make progress or that messages transit the network, i.e., even in an *asynchronous* system. Eliminating such timing assumptions is important in systems that may come under attack, where “assumption” equates to “vulnerability”.

A seminal result in distributed computing is that *only* read-write objects, and objects that can be implemented using them, have wait-free linearizable implementations [Herlihy91]. That is, while a read-

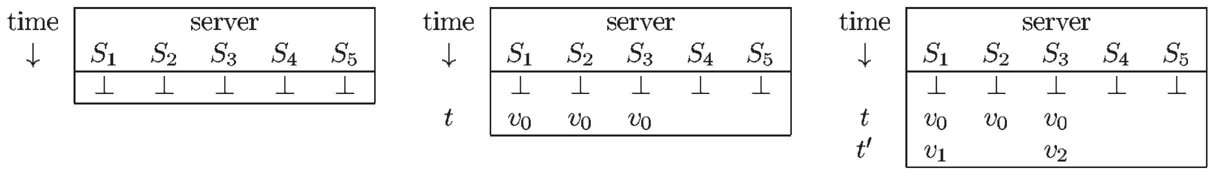
1 The client can also interact with servers indirectly through a single server, provided that the client can verify (e.g., via cryptography) that multiple servers contributed to the final result.

write storage service may have a wait-free implementation, an approach such as state machine replication, which implements linearizable operations for many other service types (e.g., read-modify-write), has no wait-free implementation. At some level this can be viewed as another expression of the well-known impossibility of deterministically live atomic broadcast [Fischer85].

Versioning. Comprehensive versioning (non-destructive updates for all operations) was introduced in the context of secure storage by our related *self-securing storage* project [Strunk00]. In this project, we implemented comprehensive versioning within a server itself, so that each write to a block creates a new version of the block without destroying the previous version. Similarly, delete operations hide the "deleted" data without actually destroying it. In our previous work, this was used to ensure that log files could not be overwritten by an attacker who compromised the computer (even the kernel) to which the storage was connected, and to permit recovering from attackers who delete or overwrite critical files. Obviously, the server cannot maintain these extra copies of blocks forever, but our empirical studies suggest that modern disks under typical workloads can keep these copies for weeks before needing to delete them. (Workloads that strive to suddenly reduce this duration are most likely indicative of an attack, and so corrective action can be taken.)

This research builds from our observation that versioning can be used as a mechanism to aid the provision of strong concurrent semantics and intrusion tolerance in a very different way. To illustrate the basic idea, consider a distributed service consisting of servers $S_1,\ldots,S_5$, each storing a replica of some data item whose value is initialized to $\perp$. The initial state of the system is shown in Figure 2.1(a). For this service to be fault tolerant, we must permit operations to complete without contacting all servers (since some servers may not respond). Suppose for the sake of illustration that a write operation completes after writing to three servers, and consider a write operation that occurs with value v_0 at time t, yielding a configuration as shown in Figure 2.1(b)

.



time	server				
↓	S_1	S_2	S_3	S_4	S_5
	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

time	server				
↓	S_1	S_2	S_3	S_4	S_5
	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
t	v_0	v_0	v_0		

time	server				
↓	S_1	S_2	S_3	S_4	S_5
	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
t	v_0	v_0	v_0		
t'	v_1		v_2		

(a) at initialization (b) at time *t* (c) at time *t'* > *t*

Figure 2.1: A data item replicated at five servers $S_1,\ldots,S_5$

A subsequent reader *R* that observes these three copies of v_0 is required to return v_0 as the written value, by the constraints of linearizability. Now suppose second and third writers begin operations to write values v_1 and v_2 at time *t'*, yielding the scenario in Figure 2.1(c). Here, the writes of v_1 and v_2 are not yet complete, either because the writers are slow or have failed. Now consider the situation that a reader *R'* sees at time *t'*. If updates are destructive, then *R'* is unable to determine which of v_0, v_1, v_2, or $\perp$ to return, as it cannot determine whether any writes ever completed. Linearizability requires that if *R* returned v_0, then *R'* cannot return $\perp$. However, if *R'* returns one of v_0, v_1, or v_2, then a subsequent reader *R''* may not even see the value *R'* returns when *R''* reads from four of the five servers. (Again, *R''* cannot be required to read from all servers due to the need to tolerate a server failure.)

The advantage of versioning is that writes are not destructive. In Figure 2.1(c), if versioning is used, a reader can observe the *history* of updates at each server, and so could see v_0 in addition to v_1 at S_1, and v_0 in addition to v_2 at S_3. This additional information permits *R'* to determine, e.g., that v_0 was a complete write if it also observes S_2, or at least that v_0 *could have been* a complete write even if *R'* cannot read S_2, ensuring that R' will not look to writes before v_0 to return as the result of this read (which would violate linearizability).

The above simple example is intended only to provide intuition as to why versioning can assist in implementing linearizable distributed services. At a high level, it permits clients to "look into the past" to determine whether a write of a value completed. Moreover, it appears that versioning also provides the ability to tolerate Byzantine servers and clients in a similar way. For example, if in Figure 2.1(c), server S_1 is corrupt and "manufactures" the value v_1 that was never part of a write operation, then v_1 will not appear in the histories of any correct servers. Similarly, if a corrupted client writes different values to different servers as part of the same write operation, readers may be able to look "behind" this information to return a value that had previously been written properly. These properties will provide the basis for an algorithm that discards values based on the number of servers' histories in which they appear.

Versioning permits us to reduce the per-request overhead on servers, and thus improve the scalability of the system. The example above demonstrates that versioning provides a basis on which readers can interpret histories of written values *without the service ordering all write operations as they are submitted*. In particular, Figure 2.1(c) shows that even with concurrent writes of v_1 and v_2, and with servers processing these writes in a different order, versioning provides a basis by which a client can determine a proper return value. A central part of this work was to develop these intuitions into a linearizable, intrusion-tolerant and scalable implementation of read-write data, and to extend these ideas to develop protocols for richer objects.

We note that if a reader can make a (correct) determination of what value to return simply by reading backward in the servers' histories, then this is a sound base on which to build a wait-free protocol. Each server's history is finite, and so the client protocol will complete in finitely many steps, regardless of the depth into servers' histories it reads to determine the proper return value.

Of course, in practice, servers will not be able to store versions forever, and so an important practical aspect of this work was managing garbage collection, i.e., pruning servers' histories. As already described, our previous research on versioning indicates that for typical file system workloads and with modern disk capacities, it is generally possible to provision a server so as to delay deletion for weeks. As such, we have significant room to work in integrating garbage collection into the system.

Quorum systems. Another platform for the advances we pursued here is novel quorum systems. Intuitively, quorum systems enable clients to complete operations on shared data objects after interacting with only a typically small subset (quorum) of servers, with no centralized locking or management, and with no server-to-server interaction. Quorums can be surprisingly small, e.g., comprised of only $O(\sqrt{n})$ servers out of a total of n servers, and thus client access protocols can efficiently scale to hundreds and possibly even thousands of servers.

More specifically, given a universe of servers U, a quorum system $Q \subseteq 2^U$ is a set of subsets (quorums) of servers with the property that any two quorums intersect. Intuitively, this property can be used to ensure that consistency is preserved across multiple operations performed at different quorums. For example, supposing only benign failures for the moment, if a client reads data at a quorum of servers, then because there is a server in that quorum that also received the last-written value for the data, the client will be sure to obtain it. Of course, in our setting, simply requiring a non-empty intersection between two quorums may not suffice, since all servers in that intersection may be corrupt.

To solve this problem, we pioneered the study of quorum systems in the context of arbitrarily (Byzantine) faulty clients and servers. Intuitively, a quorum system tolerant of Byzantine failures is a collection of subsets of servers, each pair of which intersects in a set containing sufficiently many correct servers to guarantee consistency of the replicated data as seen by correct clients. A simple example motivates the definition of Byzantine quorum systems. Suppose that all servers store a copy of a variable x, and that there is some (unknown) set B of faulty servers. First a client correctly updates the value of x at a quorum Q_1 of servers, and subsequently another client reads the value of x at the quorum Q_2. In order for the second client to get the correct, latest value of x, it is necessary for that client to distinguish the value returned by the correct servers in $Q_1 \cap Q_2$ (i.e., the servers in $Q_1 \cap Q_2 \setminus B$) from the values returned by the faulty servers and out-of-date correct servers. Supposing that at most a known threshold of b servers are faulty, i.e., $|B| \leq b$, the client can distinguish the correct value if $|Q_1 \cap Q_2| \geq 2b+1$. In particular, this condition implies that the correct value is returned by at least $b+1$ servers, and so the client can safely discard any value returned by b or fewer servers. Then, from among the remaining values (each of which was returned by at least one correct server), the client can, for example, identify the latest one using a logical timestamp protocol.

Byzantine quorum systems can offer surprising load balancing capability. Formally, this capability is captured via a measure called *load* [Naor98]. Let an *access strategy* for a quorum system $\mathcal{Q}$ be a probability distribution on the quorums in $\mathcal{Q}$. Intuitively, for each $Q \in \mathcal{Q}$, the access strategy gives the probability that a client chooses Q in order to access the service. In [Malkhi00b], we explore the load of each type of quorum system, where the load of a quorum system is the minimal access probability of the busiest server, minimizing over all access strategies. Load is a formal measure of scalability for quorum systems, in that a quorum system with lower load can provide better throughput than a quorum system with higher load. In [Malkhi00b], we present b-masking quorum systems with a load of only $O(\sqrt{b/n})$, which meets the lower bound for the load of Byzantine quorum systems [Malkhi00b]. A simple example is the M-Grid construction shown in Figure 2.2, where servers are arranged in a logical $\sqrt{n} \times \sqrt{n}$ grid, and a quorum consists of $\sqrt{b+1}$ rows and $\sqrt{b+1}$ columns.

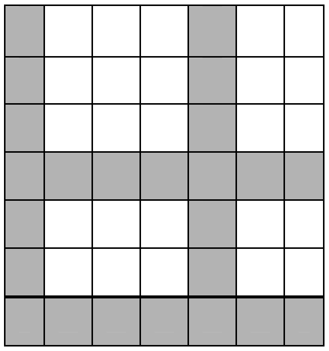


Figure 2.2: The M-grid construction, $n = 7 \times 7$, $b = 3$, with one quorum shaded.

In [Malkhi00b], we further distinguish between masking Byzantine faults and surviving a possibly larger number of benign faults, and provide masking quorum systems for this hybrid model. Our systems remain available in the face of any f crashes, where f may be significantly larger than b. The parameter f is referred to as the *availability* of the system. In addition, our constructions demonstrate asymptotic optimality in two widely accepted measures of quorum systems, namely load and failure probability. The failure probability is the probability, assuming that each server crashes with some fixed probability that all quorums in the system will contain at least one crashed server, and thus will be unavailable. The failure probability is an even more refined measure of tolerance to benign failures than the availability f, as a good system will tolerate many failure configurations with more than f crashes. Two of the constructions we achieve in [Malkhi00b] use a boosting technique, which is of interest in itself, as it can transform any regular (i.e., benign fault-tolerant) quorum system into a masking quorum system for an appropriately larger system. Thus, it makes all known quorum constructions available for Byzantine environments (of appropriate sizes).

Table 2.1 provides a sample collection of Byzantine quorum systems and their properties as functions of the number n of servers, demonstrating the resilience and cost of these as tools for survivable replication.

System	**Resilience b**	**Quorum Size**
Threshold [Malkhi98a]	$b < n/4$	$3n/4$
M-Grid [Malkhi00b]	$b < \sqrt{n}/2$	$O(\sqrt{b/n})$
BoostFPP [Malkhi00b]	$b < n/4$	$O(\sqrt{b/n})$

Table 2.1: Masking quorum systems and their properties (b = bound on Byzantine failures)

Nested method invocations. In a non-distributed, object-oriented programming environment, objects are regularly embedded inside of each other. A field within one object can contain another object. Any given object can be referenced in multiple locations; it could be referenced in a field of one object, while simultaneously being nested several layers deep within another object. In the latter case, passing the embedding object as a parameter to a method may make the embedded object directly available to that method. In fact, nested objects are fundamental to the object-oriented programming paradigm.

In a distributed, object-oriented programming environment, embedded objects take on a new level of complexity, as well as an associated level of indirection. Without nesting, distributed objects of the form we envision are structured as in Figure 2.3(a). This figure shows a client that interacts with object replicas stored on servers. The client does so via a reference for the distributed object, which we call a *handle*. All distributed object interactions go through handles; in fact, with respect to the local programming environment, the handle itself is the distributed object, and provides the same method invocation interface as the object replicas. As with local objects, distributed object handles can be embedded within local objects or within the replicas of other distributed objects, resulting in *object nesting*; see Figure 2.3(b). They can be passed around as explicit parameters and as embedded fields of parameters. A single distributed object could be simultaneously referenced from a client and from the replicas of another distributed object, as demonstrated in Figure 2.3(b).

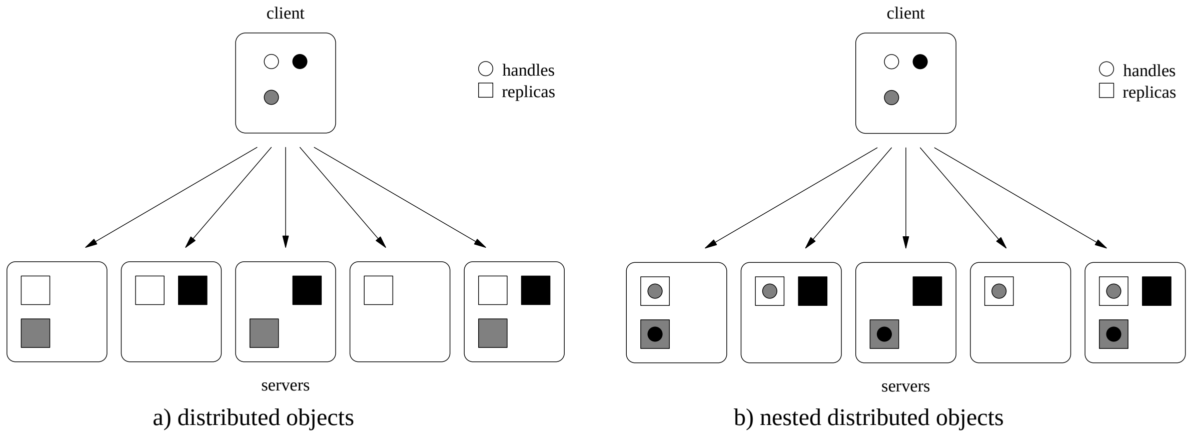


Figure 2.3: Distributed objects with nesting.

Ensuring correct method invocation semantics in the face of object nesting introduces a number of challenges. First, a method invocation by the client on (the handle of) a distributed object induces method invocations on the replicas of that object. Each of these replicas, in turn, may invoke a method on the handle of the embedded, distributed object it contains. In this scenario, the embedded ("inner") distributed object is therefore invoked multiple times, for the same method invocation on the "outer" distributed object; if these invocations are not idempotent, then the state of the inner distributed object may be corrupted, and at the very least, the transparency of object distribution will suffer. Second, to support nested method invocations, it is necessary to grant authority to each replica of the outer object to invoke methods on the inner object. In an intrusion-tolerant system that presumes that servers may be corrupted, this implies corrupted *clients* for the embedded object. Known method invocation protocols for quorum-based object replication (e.g., that used in Fleet [Malkhi01]) do not ensure consistent method invocations in the face of corrupted clients. Even if this were ensured, there is nothing to prevent a corrupt client from "correctly" invoking a method with, e.g., parameters that are incorrect from the application perspective.

Our approach to addressing these issues will be to develop an authorization framework and method invocation protocol that executes only method invocations performed by at least b+1 *client* object replicas, where b denotes the number of corruptions that the client object is designed to tolerate. We have previously developed such an approach in the context of state machine replication [Narasimhan99], but a more scalable, quorum-based approach introduces new challenges. In particular, if the client object replicas invoke different quorums of server replicas in their attempts, then it is possible that no single server replica, much less a quorum, will witness the same method invocation from sufficiently many servers to convince it to process the invocation.

2.1.2 Systems

This project utilized the techniques described in Section 2.1.1 to achieve significant advances in scalability for intrusion-tolerant systems, and address the challenges introduced by these techniques as outlined there. We experimented with implementations of the protocols that we developed, within the context of example systems extended to utilize them. Below we provide a brief overview of the two systems that will primarily serve as testbeds: PASIS and Fleet.

PASIS: PASIS is a survivable storage system developed in a project by the same name funded by DARPA IPTO. (The funding for this project ended in December 2003.) PASIS provides for the confidentiality, integrity, and availability of stored data even when some storage nodes fail or are compromised by an intruder.

It does so by encoding and distributing data across storage nodes. There are many data distribution schemes applicable to survivable storage, including replication, striping, erasure-resilient coding, secret sharing, and various combinations (including combinations with encryption). No single data distribution scheme is right for all systems, and a key focus of the PASIS project has been developing an engineering understanding of the trade-offs involved with the decision of how data is encoded and distributed [Wylie01].

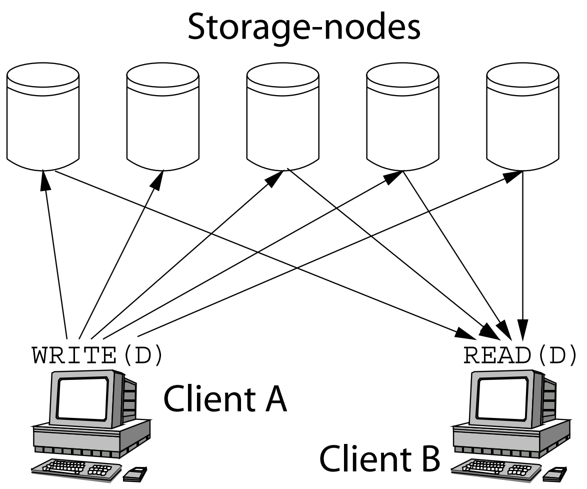


Figure 2.4: High-level architecture of PASIS survivable storage. Spreading data redundantly across storage-nodes improves its fault-tolerance. Clients write and read data from multiple storage nodes.

Figure 2.4 illustrates the abstract architecture of PASIS. To write a data-block *D*, Client *A* issues write requests to storage-nodes, with replicas or shares (when *m*-of-*n* erasure coding is employed). To read *D*, Client *B* issues read requests to storage-nodes to ensure the latest update is seen and to fetch sufficient shares. This scheme provides access to data-blocks even when subsets of the storage-nodes have failed.

To provide reasonable semantics, a storage system must guarantee that readers see consistent data-block values. Specifically, the linearizability [Herlihy90] of operations is desirable for a shared storage system. The early PASIS prototypes required that clients coordinate concurrent accesses explicitly (e.g., via locking, as would occur in a distributed database system).

Within the context of PASIS, an initial exploration was begun into the viability of using versioning storage-nodes [Strunk00, Soules02] as a foundation for efficient consistency, even with concurrent readers and writers. As described in Section 2.1.1, if each storage-node keeps all versions of its data-blocks, it becomes possible to achieve useful consistency semantics, and to do so without excess communication or I/O in the common cases of non-failed clients and minimal write sharing. Such versioning is also useful in storage systems because it simplifies several aspects of data protection, including recovery from user mistakes [Santry99], recovery from system failure [Hitz94], and survival of client compromises [Strunk00]. Further, it can be implemented with minimal performance cost ($\approx$2%) [Strunk00] and capacity demand ($\approx$10%) [Soules02].

Our early results suggested great promise in the approach, and led to development and further exploration in combination with the Byzantine quorum concept introduced by Fleet.

Fleet: Fleet is a distributed object system funded through DARPA ATO. (The funding for this project ended in June 2004.) Fleet provides the capability to transform an arbitrary serializable Java object into a remote one that lives beyond the JVM in which it was created, and that can be invoked by clients outside that JVM. In the process of making the object remote, Fleet distributes the object to multiple different servers; we call this a *distributed object*. By default, these replicas of a distributed object are managed through a quorum-based method invocation protocol that enforces linearizability while masking the Byzantine failure of a limited number of object replicas [Chockler01].

The architecture of a distributed object in Fleet is essentially that shown in Figure 2.3(a). When an object is distributed, it is replaced in its creating JVM by a *handle* for the object that implements the same interface as the original object. The replicas are instantiated on the remote servers. Subsequent method invocations on the handle are converted into method invocations on a quorum of replicas, managed (again, by default) via a method invocation protocol [Chockler01]. In addition, other clients can instantiate handles for this object, either by discovering its replicas at the servers and instantiating a handle for it, or being passed a handle explicitly. However, the original version of Fleet does not support nested objects and nested method invocations [Malkhi01], and as such does not accommodate circumstances as depicted in Figure 2.3(b). In particular, the default method invocation protocol of Fleet [Chockler01] does not accommodate corrupt clients, and we are aware of no quorum-based protocol that does so. We filled this gap in the proposed work.

2.1.3 Comparison with Pre-Existing Technology

Most prior systems implementing Byzantine fault-tolerant services adopt the state machine approach (for a tutorial, see [Schneider90]), whereby all operations are processed by server replicas in the same order (*atomic broadcast*). While this approach supports a linearizable, Byzantine fault-tolerant implementation of *any* deterministic object, it has a number of limitations that we strive to overcome in this proposed work. First and foremost, it scales poorly both in the number of client requests and in the number of servers. It scales poorly in the number of client requests because it does not offer opportunities for load dispersion among servers; all servers process every request. It scales poorly in the number of servers because adding more servers increases the number of servers among which ordering must occur.

A second limitation of state machine replication, when considering it to support read-write data storage, is that it cannot provide a wait-free service. That is, due to its generality, state machine replication is constrained by fundamental impossibility results [Fischer85, Herlihy91, Jayanti98] that prohibit this strong liveness property, even though the requirements of read-write storage do not mandate this limitation. Instead, such systems achieve liveness only under stronger timing assumptions, such as synchrony (e.g., [Pittelli89, Shrivastava92, Cristian95]) or partial synchrony [Dwork88] (e.g., [Reiter94, Kihlstrom01, Castro02]), or probabilistically (e.g., [Cachin01]). That is, the generality of state machine replication is, in fact, detrimental to the correctness properties that can be offered for read-write storage, by inducing additional assumptions to gain liveness. This is a second good reason to depart from state machine replication where possible.

As we have discussed, a more scalable alternative to replicated state machines is Byzantine quorum systems [Malkhi98a]. Protocols for supporting a linearizable implementation of any deterministic object using Byzantine quorums have been developed (e.g., [Chockler01]), but also necessarily forsake wait-freedom to do so. For the specific case of read-write data, Byzantine fault-tolerant protocols for implementing read-write access using quorums are described in [Herlihy87, Malkhi98b, Pierce01, Martin02]. Of these, only [Martin02] achieves linearizability in our fault model, and this work is also closest to ours in that it uses a variation on versioning (very abstractly, where clients listen for future versions, as opposed to read past versions as we propose here). More significantly, however, in our fault model, specifically including faulty clients, [Martin02] has the disadvantages that it does not work for erasure-coded data; (ii) it suffers the exchange of n^2 messages among the n servers per write, and so does not scale well; and (iii) it requires significant computation by requiring digital signatures by clients. As such, it leaves very significant room for improvement; we will achieve that improvement.

We contrast our use of versioning for concurrency control with systems in which each write creates a new, immutable version of a data item to which subsequent reads are directed (e.g., [Reed80, Reed83, Mullender85]). This approach shifted the concurrency control problem to the metadata mechanism that resolves the data-item name to a version. So, systems that employ this approach (e.g., Past [Rowstron01], CFS [Dabek01], Farsite [Adya02] and the archival portion of Oceanstore [Kubiatowicz00]) require a separate concurrency control mechanism to manage this metadata (e.g., state machine replication, as implemented in BFT [Castro02] in the case of Farsite and Oceanstore).

The only prior system of which we are aware that supported embedded, distributed objects, and nested invocations by one object on another, in an intrusion-tolerant way is our own Immune system

[Narasimhan99]. However, this approach utilized state machine replication to implement objects generically, and so suffers from the same scaling problems detailed above. As described in Section 2.1.1, we dramatically advanced the scalability of this class of system, but faced a number of challenges in doing so as we moved to a quorum-based and potentially version-based approach. The payoff, of course, is a dramatically more scalable basis for building services of almost unlimited types.

3 THE READ WRITE CONSISTENCY PROTOCOL AND LAZY VERIFICATION IN BYZANTINE FAULT-TOLERANT DISTRIBUTED STORAGE SYSTEMS

3.1 Introduction

Survivable storage systems spread data redundantly across a set of distributed storage-nodes in an effort to ensure its availability despite the failure or compromise of storage-nodes. Some systems support only replication, while others also use more space-efficient (and network-efficient) erasure-coding approaches in which a *fragment*, which is smaller than a full copy, is stored at each storage-node. Client read and write operations interact with multiple storage-nodes, according to some consistency protocol, to implement consistency in the face of client and storage-node faults and concurrent operations.

This chapter describes and evaluates a new consistency protocol that operates in an asynchronous environment and tolerates Byzantine failures of clients and storage-nodes. It then goes on to introduce *lazy verification* and describe implementation techniques for exploiting its potential. Lazy verification enables the system to amortize verification effort over multiple operations, to perform verification during otherwise idle time, and to have only a sub-set of storage-nodes perform verification.

3.1.1 The Consistency Protocol

The consistency protocol supports a hybrid failure model in which up to t storage-nodes may fail: $b \leq t$ of these failures can be Byzantine and the remainder can be crash. The protocol also supports the use of m-of-n erasure codes (i.e., m-of-n fragments are needed to reconstruct the data), which usually require less network bandwidth (and storage space) than full replication.

Briefly, the protocol works as follows. To perform a write, a client erasure codes a data-item into a set of fragments, determines the current logical time and then writes the time-stamped fragments to at least a threshold set of storage-nodes. Storage-nodes keep all versions of the fragments they are sent (in practice, until garbage collection frees them). To perform a read, a client fetches the latest fragment versions from a subset of storage-nodes and determines whether they comprise a completed write; usually, they do. If they do not, additional and historical fragments are fetched, and repair may be performed, until a completed write is observed. The fragments are then decoded and the data-item is returned.

The protocol gains efficiency from five features. First, it supports the use of space-efficient m-of-n erasure codes that can substantially reduce communication overheads. Second, most read operations complete in a single round trip: reads that observe write concurrency or failures (of storage-nodes or a client write) may incur additional work. Most studies of distributed storage systems (e.g., [Baker91, Noble94]) indicate that concurrency is uncommon (i.e., writer-writer and writer-reader sharing occurs in well under 1% of operations). Failures, although tolerated, ought to be rare. Third, incomplete writes are replaced by subsequent writes or reads (that perform repair), thus preventing future reads from incurring any additional cost; when subsequent writes do the fixing, additional overheads are never incurred. Fourth, most protocol processing is performed by clients, increasing scalability via the well-known principle of shifting work from servers to clients [Howard88]. Fifth, the protocol only requires the use of cryptographic hashes, rather than more expensive digital signatures.

Following sections describe the protocol in detail and develop bounds for thresholds in terms of the number of failures tolerated (i.e., the protocol requires at least $2t+2b+1$ storage-nodes). Also described and evaluated is its use in a prototype storage system called PASIS [Wylie00]. To demonstrate that our protocol is efficient in practice, we compare its performance to BFT [Castro01, Castro02], the Byzantine fault-tolerant replicated state machine implementation that Castro and Rodrigues have made available [Castro]. Experiments show that PASIS scales better than BFT in terms of server network utilization and in terms of work performed by the server. Experiments also show that response times of BFT (using multicast) and PASIS are comparable.

This protocol is timely because many research storage systems are investigating practical means of achieving high fault tolerance and scalability of which some are considering the support of erasure-coded data (e.g., [Frolund03, Ganger03, Kubiatowicz00, Morris02]). Our protocol for Byzantine-tolerant erasure-coded storage can provide an efficient, scalable, highly fault-tolerant foundation for such storage systems.

3.1.2 Lazy Verification

Sections 3.7 through 3.11 of this chapter describe a read/write consistency protocol that uses versioning storage to avoid proactive write-time verification. Rather than verification occurring during every write operation, it is performed by clients during read operations. Read-time verification eliminates the work for writes that become obsolete before being read, such as occurs when the data is deleted or overwritten. Such data obsolescence is quite common in storage systems with large client caches, as most reads are satisfied by the cache but writes must be sent to storage-nodes to survive failures. One major downside to read-time verification, however, is the potentially unbounded cost: a client may have to sift through many ill-formed or incomplete write values. A Byzantine client could degrade service by submitting large numbers of bad values. A second practical issue is that our particular approach to verification for erasure-coding, called validating timestamps, requires a computation cost equivalent to fragment generation on every read.

Lazy verification operates by having the storage-nodes perform verification in the background. It avoids verification for data that has a short lifetime and is never read. In addition, it allows verification to occur during otherwise idle periods, which can eliminate all verification overhead in the common cases. Clients will only wait for verification if they read a data-item before verification of the most recent write operation to it completes. Our lazy verification implementation limits the number of unverified write operations in the system and, thus, the number of retrievals that a reader must perform to complete a read. When the limit on unverified writes is reached, each write operation must be verified until the storage-nodes restore some slack. Limits are tracked on a per-client basis, a per-data-item basis, and on a storage-node basis (locally).

Combined, the different limits can mitigate the impact of Byzantine clients (individually and collectively) while minimizing the impact on correct clients. In the worst case, a collection of Byzantine clients can force the system to perform verification on every write operation, which is the normal-case operation for most other protocols.

This section describes and evaluates the design and implementation of lazy verification in the PASIS storage system [Goodson04a]. Several techniques are introduced for reducing the impact of verification on client reads and writes, as well as bounding the read-time delay that faulty clients can insert into the system. For example, a subset of updated storage-nodes can verify a write operation and notify the others, reducing the communication complexity from $O(N^2)$ to $O(bN)$ in a system of N storage-nodes tolerating b Byzantine failures; this reduces the number of messages by 33% even in the minimal system configuration. Appropriate scheduling allows verification to complete in the background. Indeed, in workloads with idle periods, the cost of verification is hidden from both the client read and write operations. Appropriate selection of updates to verify can maximize the number of verifications avoided (for writes that become obsolete). The overall effect is that, even in workloads without idle periods, the lazy verification techniques implemented provide over a factor of four higher write throughput when compared to a conventional approach that proactively performs verification at write-time.

3.2 Efficient Byzantine-tolerant Erasure-coded Storage: Background

In a fault-tolerant, or survivable, distributed storage system, clients write and (usually) read data from multiple storage-nodes. This scheme provides access to data-items even when subsets of the storage-nodes have failed. One difficulty created by this architecture is the need for a consistent view, across storage-nodes, of the most recent update. Without such consistency, data loss is possible. To provide reasonable semantics, storage systems must guarantee that readers see consistent data-item values. Specifically, the linearizability [Herlihy90] of operations is desirable for a shared storage system.

A common data distribution scheme used in such systems is replication. Since each storage-node has a complete instance of the data-item, the main difficulty is identifying and retaining the most recent instance. Alternately, more space-efficient erasure coding schemes can be used to reduce network load and storage consumption. With erasure coding schemes, reads require fragments from multiple servers. Moreover, to decode the data-item, the set of fragments read must correspond to the same write operation.

Most prior systems implementing Byzantine fault tolerant services adopt the replicated state machine approach [Schneider90], whereby all operations are processed by server replicas in the same order (*atomic broadcast*). While this approach supports a linearizable, Byzantine fault-tolerant implementation of any deterministic object, such an approach cannot be wait-free [Fischer85, Herlihy91, Jayanti98]. Instead, such systems achieve liveness only under stronger timing assumptions. An alternative to state machine replication is a Byzantine quorum system [Malkhi97], from which our protocol inherits techniques (i.e., our protocol can be considered a Byzantine quorum system that uses the threshold quorum construction). Protocols for supporting a linearizable implementation of any deterministic object using Byzantine quorums have been developed (e.g., [Malkhi01]), but also necessarily forsake wait-freedom to do so. Additionally, most protocols accessing Byzantine quorum systems utilize computationally expensive digital signatures.

Byzantine fault-tolerant protocols for implementing read-write objects using quorums are described in [Herlihy87, Malkhi97, Martin02]. In our protocol, a reader may retrieve fragments for several versions of the data-item in the course of identifying the return value of a read. Similarly, readers in [Martin02] “listen” for updates (versions) from storage-nodes until a complete write is observed. Conceptually, our approach differs in that clients read past versions, versus listening for future versions broadcast by servers. In our fault model, especially in consideration of faulty clients, our protocol has several advantages. First, our protocol works for erasure-coded data, whereas extending [Martin02] to erasure coded data appears nontrivial. Second, ours provides better message efficiency. Third, ours requires less computation, in that we do not require the use of expensive digital signatures. Advantages of [Martin02] are that it tolerates a higher fraction of faulty servers than our protocol, and does not require servers to store a potentially unbounded number of data-item versions. Our prior analysis of versioning storage, however, suggests that the latter is a non-issue in practice [Strunk00], and even under attack this can be managed using the garbage collection mechanism we describe in Section 3.5.1.1.

3.3 System Model

We describe the system infrastructure in terms of *clients* and *storage-nodes*. There are N storage-nodes and an arbitrary number of clients in the system. A client or storage-node is *correct* in an execution if it satisfies its specification throughout the execution. A client or storage-node that deviates from its specification *fails*. We assume a hybrid failure model for storage-nodes. Up to t storage-nodes may fail, $b \leq t$ of which may be Byzantine faults [Lamport82]; the remainder are assumed to crash. We make no assumptions about the behavior of Byzantine storage-nodes and Byzantine clients. A client or storage-node that does not exhibit a Byzantine failure (it is either correct or crashes) is *benign*.

Our protocol tolerates Byzantine faults in any number of clients and a limited number of storage nodes while implementing linearizable and wait-free read-write objects. Linearizability is adapted appropriately for Byzantine clients (we discuss the necessary adaptations in [Goodson04b]). Wait-freedom functions as in the model of Jayanti et al. [Jayanti98] and assumes no storage exhaustion.

The protocol tolerates crash and Byzantine clients. As in any practical storage system, an authorized Byzantine client can write arbitrary values to storage, which affects the value of the data, but not its consistency. We assume that Byzantine clients and storage-nodes are computationally bounded so that we can benefit from cryptographic primitives.

We assume an asynchronous model of time (i.e., we make no assumptions about message transmission delays or the execution rates of clients and storage-nodes, except that it is non-zero). We assume that communication between a client and a storage-node is point-to-point, reliable, and authenticated: a correct storage-node (client) receives a message from a correct client (storage-node) if and only if that client (storage-node) sent it.

There are two types of operations in the protocol — *read operations* and *write operations* — both of which operate on data-items. Clients perform read/write operations that issue multiple read/write requests to storage-nodes. A read/write request operates on a *data-fragment*. A data-item is *encoded* into data-fragments. Clients may encode data items in an erasure-tolerant manner; thus the distinction between data-items and data-fragments. Requests are *executed* by storage-nodes; a correct storage-node that executes a write request *hosts* that write operation.

Storage-nodes provide fine-grained versioning; correct storage-nodes host a version of the data-fragment for each write request they execute. There is a well known zero time, **0**, and null value, $\perp$, which storage-nodes can return in response to read requests. Implicitly, all stored data is initialized to $\perp$ at time **0**.

3.4 Byzantine Fault-tolerant Consistency Protocol

This section describes our Byzantine fault-tolerant consistency protocol, which efficiently supports erasure-coded data-items by taking advantage of versioning storage-nodes. It presents the mechanisms employed to encode and decode data, and to protect data integrity from Byzantine storage nodes and clients. It then describes, in detail, the protocol in pseudo-code form. Finally, it develops constraints on protocol parameters to ensure linearizability and wait-freedom.

3.4.1 Overview

At a high level, the protocol proceeds as follows. Logical timestamps are used to totally order all write operations and to identify data-fragments pertaining to the same write operation across the set of storage-nodes. Data-fragments are generated by erasure-coding data-items. For each write, a logical timestamp is constructed by the client that is guaranteed to be unique and greater than that of the *latest complete write* (the complete write with the highest timestamp). This is accomplished by querying storage-nodes for the greatest timestamp they host, and then incrementing the greatest response. In order to verify the integrity of the data, a hash that can verify data-fragment correctness is appended to the logical timestamp.

To perform a read operation, clients issue read requests to a subset of storage-nodes. Once at least a read threshold of storage-nodes reply, the client identifies the *candidate*—the response with the greatest logical timestamp. The set of read responses that share the timestamp of the candidate comprise the *candidate set*. The read operation classifies the candidate as *complete*, *repairable*, or *incomplete*. If the candidate is classified as complete, the data-fragments, timestamp, and return value are validated. If validation is successful, the candidate's value is decoded and returned; the read operation is complete. Otherwise, the candidate is reclassified as incomplete. If the candidate is classified as repairable, it is repaired by writing data-fragments back to the original set of storage-nodes. Prior to performing repair, data-fragments are validated in the same manner as for a complete candidate. If the candidate is classified as incomplete, the candidate is discarded, previous data-fragment versions are requested, and classification begins anew. All candidates fall into one of the three classifications, even those corresponding to concurrent or failed write operations.

3.4.2 Mechanisms

Several mechanisms are used in our protocol to achieve linearizability in the presence of Byzantine faults.

3.4.2.1 Erasure Codes

We use threshold erasure coding schemes, in which *N* data-fragments are generated during a write (one for each storage-node), and any *m* of those data-fragments can be used to decode the original data-item. Moreover, any *m* of the data-fragments can deterministically generate the other *N* – *m* data-fragments.

3.4.2.2 Data-fragment Integrity

Byzantine storage-nodes can corrupt their data-fragments. As such, it must be possible to detect and mask up to *b* storage-node integrity faults.

Cross checksums: Cross checksums [Gong89] are used to detect corrupt data-fragments. A cryptographic hash of each data-fragment is computed. The set of *N* hashes are concatenated to form the cross checksum of the data-item. The cross checksum is stored with each data-fragment (i.e., it is replicated *N* times). Cross checksums enable read operations to detect data-fragments that have been modified by storage-nodes.

3.4.2.3 Write Operation Integrity

Mechanisms are required to prevent Byzantine clients from performing a write operation that lacks integrity. If a Byzantine client generates random data-fragments (rather than erasure coding a data-item correctly), then each of the $\binom{N}{m}$ permutations of data-fragments could "recover" a distinct data-item. These attacks are similar to *poisonous writes* for replication as described by Martin et al. [Martin02]. To protect against Byzantine clients, the protocol must ensure that read operations only return values that are written correctly (i.e., that are *single-valued*). To achieve this, the cross checksum mechanism is extended in three ways: validating timestamps, storage-node verification, and validated cross checksums.

Validating timestamps: To ensure that only a single set of data-fragments can be written at any logical time, the hash of the cross checksum is placed in the low order bits of the logical timestamp. Note, the hash is used for space-efficiency; instead, the entire cross checksum could be placed in the low bits of the timestamp.

Storage-node verification: On a write, each storage-node verifies its data-fragment against its hash in the cross checksum. The storage-node also verifies the cross checksum against the hash in the timestamp. A correct storage-node only executes write requests for which both checks pass. Via this approach, a Byzantine client cannot make a correct storage-node appear Byzantine. It follows that only Byzantine storage-nodes can return data-fragments that do not verify against the cross checksum.

Validated cross checksums: To ensure that the client that performed a write operation acted correctly, the reader must validate the cross checksum. To validate the cross checksum, all *N* data-fragments are required. Given any *m* data-fragments, the full set of *N* data-fragments a correct client should have written can be generated. The "correct" cross checksum can then be computed from the regenerated set of data-fragments. If the generated cross checksum does not match the verified cross checksum, then a Byzantine client performed the write operation. Only a single-valued write operation can generate a cross checksum that verifies against the validating timestamp.

3.4.2.4 Authentication

Clients and storage-nodes must be able to validate the authenticity of messages. We use an authentication scheme based on pair-wise shared secrets (e.g., between clients and storage-nodes), in which RPC arguments and replies are accompanied by an HMAC [Bellare96] (using the shared secret as the key). We assume an infrastructure is in place to distribute shared secrets. Our implementation uses an existing Kerberos [Steiner88] infrastructure.

3.4.3 Pseudo-code

The pseudo-code for the protocol is shown in Figures 3.1 and 3.2. The symbol *LT* denotes logical time and $LT_{\text{candidate}}$ denotes the logical time of the candidate. The set $\{D_1, \ldots, D_N\}$ denotes the *N* data-fragments; likewise, $\{S_1, \ldots, S_N\}$ denotes the set of *N* storage-nodes. In the pseudo-code, the binary operator '|' denotes string concatenation. Simplicity and clarity in the presentation of the pseudo-code was chosen over obvious optimizations that are in the actual implementation.

```
WRITE(Data) :
1: Time:= READ_TIMESTAMP()
2: {$D_1, . . . ,D_N$}:= ENCODE(Data)
3: CC := MAKE_CROSS_CHECKSUM({$D_1, . . . ,D_N$})
4: LT := MAKE_TIMESTAMP(Time, CC)
5: DO_WRITE({$D_1, . . . ,D_N$}, LT, CC)

READ_TIMESTAMP() :
1: for all $S_i \in \{S_1, . . . ,S_N\}$ do
2:     SEND($S_i$ , TIME_REQUEST)
3: end for
4: ResponseSet := Ø
5: repeat
6:     ResponseSet := ResponseSet $\cup$ RECEIVE(S, TIME_RESPONSE)
7: until (UNIQUE_SERVERS(ResponseSet) = $N - t$)
8: Time := MAX[ResponseSet.LT.Time]
9: RETURN(Time)

MAKE_CROSS_CHECKSUM({$D_1, . . . ,D_N$}) :
1: for all $D_i \in \{D_1, . . . ,D_N\}$ do
2:     $H_i$ := HASH($D_i$ )
3: end for
4: CC := $H_1$| . . . |$H_N$
5: RETURN(CC)

MAKE_TIMESTAMP($LT_{max}$, CC) :
1: LT:Time := $LT_{max}$·Time + 1
2: LT:Verifier := HASH(CC)
3: RETURN(LT)

DO_WRITE({$D_1, . . . ,D_N$}, LT, CC) :
1: for all $S_i \in \{S_1, . . . ,S_N\}$ do
2:     SEND($S_i$, WRITE_REQUEST, LT, $D_i$, CC)
3: end for
4: ResponseSet := Ø
5: repeat
6:     ResponseSet := ResponseSet $\cup$ RECEIVE(S, WRITE_RESPONSE)
7: until (UNIQUE_SERVERS(ResponseSet) = $N - t$)
```

Figure 3.1: Write operation pseudo-code.

3.4.3.1 Storage-node Interface

Storage-nodes offer interfaces to: write a data-fragment with a specific logical time (`WRITE`); query the greatest logical time of a hosted data fragment (`TIME_REQUEST`); read the hosted data-fragment with the greatest logical time (`READ_LATEST`); and read the hosted data-fragment with the greatest logical time at or before some logical time (`READ_PREV`). Due to its simplicity, storage-node pseudo-code is omitted.

3.4.3.2 Write Operation

The `WRITE` operation consists of determining the greatest logical timestamp, constructing write requests, and issuing the requests to the storage-nodes. First, a timestamp greater than, or equal to, that of the latest complete write must be determined. Collecting $N - t$ responses, on line 7 of `READ_TIMESTAMP`, ensures that the response set intersects a complete write at a correct storage-node. In practice, the timestamp of the latest complete write can be observed with fewer responses.

Next, the `ENCODE` function, on line 2 of `WRITE`, encodes the data-item into N data-fragments. The data-fragments are used to construct a cross checksum from the concatenation of the hash of each data-fragment (line 3). The function `MAKE_TIMESTAMP`, called on line 4, generates a logical timestamp to be used for the current write operation. This is done by incrementing the high order bits of the greatest observed logical timestamp from the *ResponseSet* (i.e., *LT:TIME*) and appending the *Verifier*. The *Verifier* is just the hash of the cross checksum.

Finally, write requests are issued to all storage-nodes. Each storage-node is sent a specific data-fragment, the logical timestamp, and the cross checksum. A storage-node validates the cross checksum with the verifier and validates the data-fragment with the cross checksum before executing a write request (i.e., storage-nodes call `VALIDATE` listed in the read operation pseudo-code). The write operation returns to the issuing client once `WRITE_RESPONSE` messages are received from $N - t$ storage-nodes (line 7 of `DO_WRITE`). Since the environment is asynchronous, a client can wait for no more than $N - t$ responses. The function `UNIQUE_SERVERS` determines how many unique servers are present in the candidate set; it ensures that only a single response from each Byzantine storage-node is counted.

3.4.3.3 Read Operation

The read operation iteratively identifies and classifies candidates, until a repairable or complete candidate is found. Once a repairable or complete candidate is found, the read operation validates its correctness and returns the data.

The read operation begins by issuing READ_LATEST requests to all storage-nodes (via the DO_READ function). Each storage-node responds with the data-fragment, logical timestamp, and cross checksum corresponding to the greatest timestamp it has executed. The integrity of each response is individually validated through the VALIDATE function, called on line 7 of DO_READ. This function checks the cross checksum against the *Verifier* found in the logical timestamp and the data-fragment against the appropriate hash in the cross checksum. Although not shown in the pseudocode, the client only considers responses from storage nodes to READ_PREV requests that have timestamps strictly less than that given in the request.

Since, in an asynchronous system, slow storage-nodes cannot be differentiated from crashed storage-nodes, only $N - t$ read responses can be collected (line 10 of DO_READ). Since correct storage-nodes perform the same validation before executing write requests, the only responses that can fail the client's validation are those from Byzantine storage-nodes. For every discarded Byzantine storage-node response, an additional response can be awaited.

```
READ() :
1: ResponseSet := DO_READ(READ_LATEST_REQUEST, ?)
2: loop
3:     ⟨CandidateSet, $LT_{candidate}$⟩ :=
                    CHOOSE_CANDIDATE(ResponseSet)
4:     if (|CandidateSet| ≥ INCOMPLETE then
5:         /* Complete or repairable write found */
6:         {$D_1, \ldots, D_N$} := GENERATE_FRAGMENTS(CandidateSet)
7:         $CC_{valid}$ := MAKE_CROSS_CHECKSUM({$D_1, \ldots, D_N$})
8:         if ($CC_{valid}$ = CandidateSet.CC) then
9:             /* Cross checksum is validated */
10:            if (|CandidateSet| < COMPLETE) then
11:                /* Repair is necessary */
12:                DO_WRITE({$D_1, \ldots, D_N$}, $LT_{candidate}$, $CC_{valid}$)
13:            end if
14:            Data := DECODE({$D_1, \ldots, D_N$})
15:            RETURN(Data)
16:        end if
17:    end if
18:    /* Incomplete or cross checksum did not validate, loop again */
19:    ResponseSet := DO_READ(READ_PREV_REQUEST, $LT_{candidate}$)
20: end loop

DO READ(READ COMMAND, LT) :
1: for all Si ∈ {$S_1, \ldots, S_N$} do
2:     SEND(Si, READ_COMMAND, LT)
3: end for
4: ResponseSet := Ø
5: repeat
6:     Resp := RECEIVE(S, READ_RESPONSE)
7:     if (VALIDATE(Resp.D, Resp.CC, Resp.LT, S) = TRUE) then
8:         ResponseSet := ResponseSet ∪ Resp
9:     end if
10: until (UNIQUE_SERVERS(ResponseSet) = N - t)
11: RETURN(ResponseSet)

VALIDATE(D, CC, LT, S) :
1: if ((HASH(CC) ≠ LT.Verifier) OR (HASH(D) ≠ CC[S])) then
2:     RETURN(FALSE)
3: end if
4: RETURN(TRUE)
```

Figure 3.2: Read operation pseudo-code

After sufficient responses have been received, a candidate for classification is chosen. The function CHOOSE_CANDIDATE, called on line 3 of READ, determines the candidate timestamp, denoted $LT_{\text{candidate}}$, which is the greatest timestamp found in the response set. All data-fragments that share $LT_{\text{candidate}}$ are identified and returned as the *CandidateSet*. The candidate set contains a set of validated data-fragments that share a common cross checksum and logical timestamp.

Once a candidate has been chosen, it is classified as complete, repairable, or incomplete based on the size of the candidate set. The definitions of INCOMPLETE and COMPLETE are given in the following subsection. If the candidate is classified as incomplete, a READ_PREV request is sent to each storage-node with its timestamp. Candidate classification begins again with the new response set. If the candidate is classified as either complete or repairable, the candidate set contains sufficient data fragments written by the client to decode the original data-item. To validate the observed write's integrity, the candidate set is used to generate a new set of data-fragments (line 6 of READ). A validated cross checksum, CC_{valid}, is computed from the generated data-fragments. The validated cross checksum is compared to the cross checksum of the candidate set (line 8 of READ). If the check fails, the candidate was written by a Byzantine client; the candidate is then reclassified as incomplete and the read operation continues.

If the check succeeds, the candidate was written by a correct client and the read enters its final phase. Note that for a candidate set classified as complete this check either succeeds or fails for all correct clients regardless of which storage-nodes are represented within the candidate set.

If necessary, repair is performed: write requests are issued with the generated data-fragments, the validated cross checksum, and the logical timestamp (line 10 of `READ`). Storage-nodes not hosting the write execute the write at the given logical time; those already hosting the write are safe to ignore it. Finally, the function `DECODE`, on line 14 of `READ`, decodes m data-fragments, returning the data-item.

3.4.4 Protocol Constraints

The symbol Q_C denotes a complete write operation: the threshold number of benign storage-nodes that must execute write requests for a write operation to be complete. To ensure that linearizability and wait-freedom are achieved, Q_C and N must be constrained with regard to b, t, and each other. As well, the parameter m, used in `DECODE`, must be constrained. We present safety and liveness proofs for the protocol and discuss the concept of linearizability in the presence of Byzantine clients in [Goodson04b].

Write completion: To ensure that a correct client can complete a write operation,

$Q_C \le N - t - b.$ (3.1)

Since slow storage-nodes cannot be differentiated from crashed storage-nodes, only $N - t$ responses can be awaited. As well, up to b responses received may be from Byzantine storage-nodes.

Read classification: To classify a candidate as complete, a candidate set of at least Q_C benign storage-nodes must be observed. In the worst case, at most b members of the candidate set may be Byzantine, thus,

$|CandidateSet| - b - Q_C$, so `COMPLETE` $= Q_C + b.$ (3.2)

To classify a candidate as incomplete a client must determine that a complete write does not exist in the system (i.e., fewer than Q_C benign storage-nodes host the write). For this to be the case, the client must have queried all possible storage-nodes (N_t), and must assume that nodes not queried host the candidate in consideration. So,

$|CandidateSet| + t < Q_C$, so `INCOMPLETE` $= Q_C - \mathrm{t}.$ (3.3)

Real repairable candidates: To ensure that Byzantine storage-nodes cannot fabricate a repairable candidate, a candidate set of size b must be classifiable as incomplete. Substituting b into (3),

$b + t < Q_C..$ (3.4)

Decodable repairable candidates: Any repairable candidate must be decodable. The lower bound on candidate sets that are repairable follows from (3) (since the upper bound on classifying a candidate as incomplete coincides with the lower bound on repairable):

$1 \le m \le Q_C - t.$ (3.5)

Constraint summary:

$$t + b + 1 \le Q_C \le N - t - b;$$
$$2t + 2b + 1 \le N;$$
$$1 \le m \le Q_C - t.$$

3.5 Evaluation

This section evaluates the performance and scalability of the consistency protocol in the context of a prototype storage system called PASIS [Wylie00]. We compare the PASIS implementation of our protocol with the BFT library implementation [Castro] of Byzantine fault-tolerant replicated state machines [Castro02], since it is generally regarded as efficient.

3.5.1 PASIS Implementation

PASIS consists of clients and storage-nodes. Storage nodes store data-fragments and their versions. Clients execute the protocol to read and write data-items.

3.5.1.1 Storage-node Implementation

PASIS storage nodes use the Comprehensive Versioning File System (CVFS) [Soules03] to retain data-fragments and their versions. CVFS uses a log-structured data organization to reduce the cost of data versioning. Experience indicates that retaining every version and performing local garbage collection comes with minimal performance cost (a few percent) and that it is feasible to retain complete version histories for several days [Soules03, Strunk00].

We extended CVFS to provide an interface for retrieving the logical timestamp of a data-fragment. Each write request contains a data-fragment, a logical timestamp, and a cross checksum. To improve performance, read responses contain a limited version history containing logical timestamps of previously executed write requests. The version history allows clients to identify and classify additional candidates without issuing extra read requests. Storage-nodes can also return read responses that contain no data other than version histories, which makes candidate classification more network-efficient.

Pruning old versions, or garbage collection, is necessary to prevent capacity exhaustion of storage-nodes. A storage node in isolation, by the nature of the protocol, cannot determine which local data-fragment versions are safe to remove. An individual storage-node can garbage collect a data-fragment version if there exists a later complete write for the corresponding data-item. Storage-nodes are able to classify writes by executing the read consistency protocol in the same manner as the client. We discuss garbage collection more fully in [Goodson03].

3.5.1.2 Client Implementation

The client module provides a block-level interface to higher level software, and uses a simple RPC interface to communicate with storage-nodes. The RPC mechanism uses TCP/IP. The client module is responsible for the execution of the consistency protocol.

Initially, read requests are issued to $Q_C + b$ storage nodes. PASIS utilizes read witnesses to make read operations more network efficient; only m of the initial requests request the data-fragment, while all request version histories. If the read responses do not yield a candidate that is classified as complete, read requests are issued to the remaining storage-nodes (and a total of up to $N - t$ responses are awaited). If the initial candidate is classified as incomplete, subsequent rounds of read requests fetch only version histories until a candidate is classified as either repairable or complete. If necessary, after classification, extra data fragments are fetched according to the candidate timestamp. Once the data-item is successfully validated and decoded, it is returned to the client.

3.5.1.3 Mechanism Implementation

We measure the space-efficiency of an erasure code in terms of *blowup*—the total amount of data stored over the size of the data-item. We use an information dispersal algorithm [Rabin89], which has a blowup of $\frac{N}{m}$. Our information dispersal implementation stripes the data-item across the first m data-fragments (i.e., each data-fragment is $\frac{1}{m}$ of the original data-item's size). These *stripe-fragments* are used to generate the *code-fragments* via polynomial interpolation within a Galois Field. Our implementation of polynomial interpolation was originally based on publicly available code [Dai-a] for information dispersal [Rabin89]. We modified the source to make use of stripe-fragments and added an implementation of Galois Fields of size 2^8 that use lookup tables for multiplication. Our implementation of cross checksums closely follows [Gong89]. We use MD5 for all hashes; thus, each cross checksum is $N \times 16$ bytes long. Note, that if very small blocks are used with a large N, then the overhead due to size of the cross checksum could be substantial.

3.5.2 Experimental Setup

We use a cluster of 20 machines to perform our experiments. Each machine is a dual 1 GHz Pentium III machine with 384 MB of memory. Storage-nodes use a 9 GB Quantum Atlas 10K as the storage device. The machines are connected through a 100 Mb switch. All machines run the Linux 2.4.20 SMP kernel. In all experiments, clients keep a single read or write operation for a random 16 KB block outstanding. Once an operation completes, a new operation is issued (there is no think time). For all experiments, the working set fits into memory and all caches are warmed up beforehand.

3.5.2.1 PASIS Configuration

Each storage-node is configured with 128 MB of data cache, and no caching is done on the clients. All experiments show results using write-back caching at the storage nodes, mimicking availability of 16 MB of non-volatile RAM. This allows us to focus experiments on the overheads introduced by the protocol and not those introduced by the disk subsystem. All messages are authenticated using HMACs; pair-wise symmetric keys are distributed prior to each experiment.

3.5.2.2 BFT configuration

Operations in BFT [Castro02] require agreement among the replicas (storage-nodes in PASIS). BFT requires $N = 3b + 1$ replicas to achieve agreement. Agreement is performed in four steps: (i) the client broadcasts requests to all replicas; (ii) the *primary* broadcasts pre-prepare messages to all replicas; (iii) all replicas broadcast prepare messages to all replicas; and, (iv) all replicas send replies back to the client and then broadcast commit messages to all other replicas. Commit messages are piggybacked on the next pre-prepare or prepare message to reduce the number of messages on the network. *Authenticators*, lists of MACs, are used to ensure that broadcast messages from clients and replicas cannot be modified by a Byzantine replica. All clients and replicas have public and private keys that enable them to exchange symmetric cryptographic keys used to create MACs. Logs of commit messages are checkpointed (garbage collected) periodically.

An optimistic fast path for read-only operations is implemented in BFT. The client broadcasts its request to all replicas. Each replica replies once all previous requests have committed. Only one replica sends the full reply (i.e., the data and digest), and the remainder just send digests that can verify the correctness of the data returned. If the replies from replicas do not agree, the client re-issues the read operation. Re-issued read operations perform agreement using the base BFT algorithm.

The BFT configuration does not store data to disk; instead it stores all data in memory and accesses it via memory offsets. For all experiments, BFT view changes are suppressed. BFT uses UDP rather than TCP. The BFT implementation defaults to using IP multicast. In our environment, like many, IP multicast broadcasts to the entire subnet, thus making it unsuitable for shared environments. We found that the BFT implementation code is fairly fragile when using IP multicast in our environment, making it necessary to disable IP multicast in some cases (where stated explicitly). The BFT implementation authenticates broadcast messages via authenticators, and point-to-point messages with MACs.

3.5.3 Mechanism Costs

Client and storage-node computation costs for operations on a 16 KB block in PASIS are listed in Table 3.1. For every read and write operation, clients perform erasure coding (i.e., they compute $N - m$ data-fragments given m data-fragments), generate a cross checksum, and generate a verifier. Recall that writes generate the first m data fragments by striping the data-item into m fragments. Similarly, reads must generate $N - m$ fragments, from the m they have, in order to verify the cross checksum. Storage-nodes validate each write request they receive. This validation requires a comparison of the data-fragment's hash to the hash within the cross checksum, and a comparison of the cross checksum's hash to the verifier within the timestamp.

All requests and responses are authenticated via HMACS. The cost of authenticating write requests, listed in the table, is very small. The cost of authenticating read requests and timestamp requests are similar.

	b=1	b=2	b=3	b=4
Erasure coding	1250	1500	1730	1990
Cross checksum	360	440	480	510
Validate	82	58	48	40
Verifier	1.6	2.3	3.6	4.3
Authenticate	1.5	1.5	2.1	2.1

Table 3.1: Computation costs in PASIS in µs.

3.5.4 Performance and Scalability

3.5.4.1 Response Time

Figure 3.3 shows the mean response time of a single request from a single client as a function of the tolerated number of storage-node failures. Due to the fragility of the BFT implementation with $b > 1$, IP multicast was disabled for BFT during this experiment. The focus in this plot is the slopes of the response time lines: the flatter the line the more scalable the protocol is with regard to the number of faults tolerated. In our environment, a key contributor to response time is network cost, which is dictated by the space-efficiency of the protocol.

Figure 3.4 breaks down the mean response times of read and write operations, from Figure 3.3, into the costs at the client, on the network, and at the storage-node for $b = 1$ and $b = 4$. Since measurements are taken at the user-level, kernel-level timings for host network protocol processing (including network system calls) are attributed to the "network" cost of the breakdowns. To understand the response time measurements and scalability of these protocols, it is important to understand these breakdowns.

PASIS has better response times than BFT for write operations due to the space-efficiency of erasure codes and the nominal amount of work storage-nodes perform to execute write requests. For $b = 4$, BFT has a blowup of 13× on the network (due to replication), whereas our protocol has a blowup of $\frac{17}{5} = 3.4$ on the network. With IP multicast the response time of the BFT write operation would improve significantly, since the client would not need to serialize 13 replicas over its link. However, IP multicast does not reduce the aggregate server network utilization of BFT—for $b = 4$, 13 replicas must be delivered.

PASIS has longer response times than BFT for read operations. This can be attributed to two main factors: First, the PASIS storage-nodes store data in a real file system; since the BFT-based block store keeps all data in memory and accesses blocks via memory offsets, it incurs almost no server storage costs. We expect that a BFT implementation with actual data storage would incur server storage costs similar to PASIS (e.g., around 0.7 ms for a write and 0.4 ms for a read operation, as is shown for PASIS with $b = 1$ in Figure 3.4). Indeed, the difference in read response time between PASIS and BFT at $b = 1$ is mostly accounted for by server storage costs. Second, for our protocol, the client computation cost grows as t increases because the cost of generating data-fragments grows as N increases. In addition to the $b = t$ case,

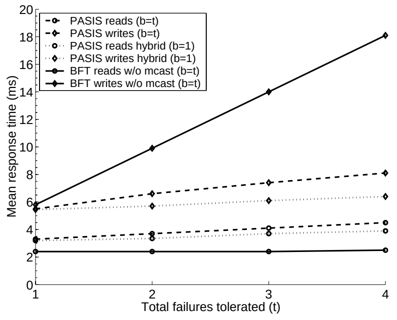


Figure 3.3: Mean Response time.

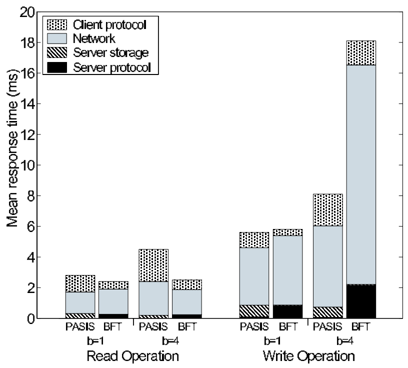


Figure 3.4: Response breakdown.

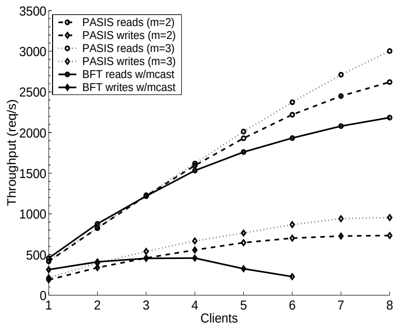


Figure 3.5: Throughput (b = 1).

Figure 3.3 shows one instance of PASIS assuming a hybrid fault model with $b = 1$. For space-efficiency, we set $m = t + 1$. Consequently, $Q_C = 2t + 1$ and $N = 3t + 2$. At $t = 1$, this configuration is identical to the Byzantine-only configuration. As t increases, this configuration is more space-efficient than the Byzantine-only configuration, since it requires $t – 1$ fewer storage-nodes. As such, the response times of read and write operations scale better.

3.5.4.2 Throughput

Figure 3.5 shows the throughput in 16 KB requests per second as a function of the number of clients (one request per client) for $b = 1$. In this experiment, BFT uses multicast, which greatly improves its network efficiency (BFT with multicast is stable for $b = 1$). PASIS was run in two configurations, one with the thresholds set to that of the minimum system with $m = 2$, $N = 5$ (write blowup of 2.5×), and one, more space-efficient, with $m = 3$, $N = 6$ (write blowup of 2×). Results show that throughput is limited by the server network bandwidth

At high load, PASIS has greater write throughput than BFT. BFT's write throughput flattens out at 456 requests per second. We observed BFT's write throughput drop off as client load increased; likewise, we observed a large increase in request retransmissions. We believe that this is due to the use of UDP and a coarse grained retransmit policy in BFT's implementation. The write throughput of PASIS flattens out at 733 requests per second, significantly outperforming BFT. This is because of the network-efficiency of PASIS. Even with multicast enabled, each BFT server link sees a full 16 KB replica, whereas each PASIS server link sees $\frac{16}{m}$ KB. Similarly, due to network space-efficiency, the PASIS configuration using $m = 3$ outperforms the minimal PASIS configuration (954 requests per second). Both PASIS and BFT have roughly the same network utilization per read operation (16 KB per operation). To be network-efficient, PASIS uses read witnesses and BFT uses "fast path" read operations. However, PASIS makes use of more storage-nodes than BFT does servers. As such, the aggregate bandwidth available for reads is greater for PASIS than for BFT, and consequently PASIS has a greater read throughput than BFT. Although BFT could add servers to increase its read throughput, doing so would not increase its write throughput (indeed, write throughput would likely drop due to the extra inter-server communication).

3.5.4.3 Scalability Summary

For PASIS and BFT, scalability is limited by either the server network utilization or server CPU utilization. Figure 3.4 shows that PASIS scales better than BFT in both. Consider write operations. Each BFT server receives an entire replica of the data, whereas each PASIS storage-node receives a data-fragment $\frac{1}{m}$ the size of a replica. The work performed by BFT servers for each write request grows with b. In PASIS, the server protocol cost decreases from 90 µs for $b = 1$ to 57 µs for $b = 4$, whereas in BFT it increases from 0.80 ms to 2.1 ms. The cost in PASIS decreases because m increases as b increases, reducing the size of the data-fragment that is validated. We believe that the server cost for BFT increases because the number of messages that must be sent increases.

3.5.5 Concurrency

To measure the effect of concurrency on the system, we measure multi-client throughput of PASIS when accessing overlapping block sets. The experiment makes use of four clients, each with four operations outstanding. Each client accesses a range of eight data blocks, with no outstanding requests from the same client going to the same block. At the highest concurrency level (all eight blocks in contention by all clients), we observed neither significant drops in bandwidth nor significant increases in mean response time. Even at this high concurrency level, the initial candidate was classified as complete 89% of the time, otherwise classification required the traversal of history information. Of these history traversals, repair was only necessary a quarter of the time (i.e., 3% of all reads required repair). Since repair occurs so seldom, the effect on response time and throughput is minimal.

3.6 Summary

We have developed an efficient Byzantine-tolerant protocol for reading and writing blocks of data. Experiments demonstrate that PASIS, a prototype storage system that uses our protocol, scales well in the number of faults tolerated, supports 60% greater write throughput than BFT, and requires significantly less server computation than BFT. Further evaluation, a proof sketch of the correctness of the protocol, and discussion of additional issues (e.g., garbage collection) can be found in the full technical report [Goodson03]. This protocol also extends into a protocol family that includes members for other system models (e.g., asynchronous or synchronous timing model, and crash-recovery failures) [Goodson04b].

3.7 Lazy Verification in Byzantine Fault-Tolerant Distributed Storage Systems: Background

This section outlines the system model on which we focus, the protocol in which we develop lazy verification, and related work.

3.7.1 System Model and Failure Types

Survivable distributed storage systems tolerate client and storage-node faults by spreading data redundantly across multiple storage-nodes. We focus on distributed storage systems that can use erasure-coding schemes, as well as replication, to tolerate Byzantine failures [Lamport82] of both clients and storage-nodes. An m-of-N erasure-coding scheme (e.g., information dispersal [Rabin89]) encodes a data-item into N fragments such that any m allows reconstruction of the original. Generally speaking, primary goals for Byzantine fault-tolerant storage systems include data integrity, system availability, and efficiency (e.g., [Cachin05b, Castro02, Goodson04a]).

Data integrity can be disrupted by faulty storage-nodes and faulty clients. First, a faulty storage-node can corrupt the fragments/replicas that it stores, which requires that clients double-check integrity during reads. Doing so is straightforward for replication, since the client can just compare the contents (or checksums) returned from multiple storage-nodes. With erasure-coding, this is insufficient, but providing all storage-nodes with the checksums of all fragments (i.e., a cross checksum [Gong89]) allows a similar approach. Second, a faulty client can corrupt data-items with *poisonous writes*. A poisonous write operation [Martin02] gives incompatible values to some of the storage-nodes; for replication, this means non-identical values, and, for erasure-coding, this means fragments not correctly generated from the original data-item (i.e., such that different subsets of m fragments will reconstruct to different values). The result of a poisonous write is that different clients may observe different values depending on which subset of storage-nodes they interact with. Verifying that a write is not poisonous is difficult with erasure-coding, because one cannot simply compare fragment contents or cross checksums—one must verify that fragments sent to storage-nodes were correctly generated from the same data-item value.

Faulty clients can also affect availability and performance by *stuttering*. A stuttering client repeatedly sends to storage-nodes a number of fragments (e.g., $m-1$ of them) that is insufficient to form a complete write operation. Such behavior can induce significant overheads because it complicates verification and may create long sequences of work that must be performed before successfully completing a read.

Our work on lazy verification occurs in the context of a protocol that operates in an asynchronous timing model. But there is no correctness connection between the timing model and verification model. Asynchrony does increase the number of storage-nodes involved in storing each data-item, which in turn increases the work involved in verification and, thus, the performance benefits of lazy verification.

3.7.2 Read/write Protocol and Delayed Verification

This section begins a description of the concept of lazy verification in the context of the PASIS read/write protocol [Goodson04a, Wylie04]. This protocol uses versioning to avoid the need to verify completeness and integrity of a write operation during its execution. Instead, such verification is performed during read operations. Lazy verification shifts the work to the background, removing it from the critical path of both read and write operations in common cases.

The PASIS read/write protocol provides linearizable [Herlihy90] read and write operations on data blocks in a distributed storage system [Goodson04b]. It tolerates crash and Byzantine failures of clients, and operates in an asynchronous timing model. Point-to-point, reliable, authenticated channels are assumed. The protocol supports a hybrid failure model for storage-nodes: up to t storage-nodes may fail, $b \leq t$ of which may be Byzantine faults; the remainder are assumed to crash. For clarity of presentation, we focus exclusively on the fully Byzantine failure model here (i.e., $b = t$), which requires $N \geq 4b + 1$ storage-nodes. The minimal system configuration ($N = 4b + 1$) can be supported only when the reconstruction threshold m satisfies $m \leq b + 1$; in our experiments we will consider $m = b + 1$ only.

An overview of the PASIS read/write protocol, including the mechanisms for erasure codes, data-fragment integrity and write-operation integrity (timestamp validation, storage-node verification, etc.) may be found in Sections 3.4.1 and 3.4.2. Under this protocol, the majority of verification work is performed by the client during the read operation. The lazy verification approach developed in the remainder of this chapter addresses this issue by having the storage-nodes communicate, prior to the next read operation if possible, to complete verification for the latest complete write. Each storage-node that observes the result of this verification can inform a client performing a read operation. If at least $b + 1$ confirm verification for the candidate, read-time verification becomes unnecessary. Section 3.8 details how lazy verification works and techniques for minimizing its performance impact.

3.7.3 Related Work

The notion of verifiability that we study is named after a similar property studied in the context of m-of-N *secret sharing*, i.e., that reconstruction from any m shares will yield the same value (e.g., [Feldman87, Pederson91]). However, to address the additional requirement of secrecy, these works employ more expensive cryptographic constructions that are efficient only for small secrets that, in particular, would be very costly if applied to blocks in a storage system. The protocols we consider here do not require secrecy, and so permit more efficient constructions. Most previous protocols perform verification proactively during write operations. When not tolerating Byzantine faults, two- or three-phase commit protocols are sufficient. For replicated data, verification can be made Byzantine fault-tolerant in many ways. For example, in the BFT system [Castro02], clients broadcast their writes to all servers, and then servers reach agreement on the hash of the written data value by exchanging messages. In other systems, such as Phalanx [Malkhi00a], an “echo” phase like that of Rampart [Reiter95] is used: clients propose a value to collect signed server echos of that value; such signatures force clients to commit the same value at all servers. The use of additional communication and digital signatures may be avoided in Byzantine fault-tolerant replica systems if replicas are *self-verifying* (i.e., if replicas are identified by a collision-resistant hash of their own contents) [Martin02].

Verification of erasure-coded data is more difficult, as described earlier. The validating timestamps of the PASIS read/write protocol, combined with read-time or lazy verification, are one approach to addressing this issue. Cachin and Tessaro [Cachin05b] recently proposed an approach based on asynchronous verifiable information dispersal (AVID) [Cachin05a]. AVID's verifiability is achieved by having each storage-node send their data fragment, when it is received, to all other storage-nodes. Such network-inefficiency reduces the benefits of erasure-coding, but avoids all read-time verification. Lazy verification goes beyond previous schemes, for both replication and erasure-coding, by separating the effort of verifying from write and read operations. Doing so allows significant reductions in the total effort (e.g., by exploiting data obsolescence and storage-node cooperation) as well as shifting the effort out of the critical path. Unrelated to content verification are client or storage-node failures (attacks) that increase the size of logical timestamps. Such a failure could significantly degrade system performance, but would not affect correctness. Bazzi and Ding recently proposed a protocol to ensure non-skipping timestamps for Byzantine fault-tolerant storage systems [Bazzi04] by using digital signatures. Cachin and Tessaro [Cachin05b] incorporate a similar scheme based on a non-interactive threshold signature scheme. Validating timestamps do not ensure non-skipping timestamps, but we believe that lazy verification could be extended to provide bounded-skipping timestamps without requiring digital signatures. This will be an interesting avenue for future work.

3.8 Lazy Verification

An ideal lazy verification mechanism has three properties: (i) it is a background task that never impacts foreground read and write operations, (ii) it verifies values of write operations before clients read them (so that clients need not perform verification), and (iii) it only verifies the value of write operations that clients actually read. This sec- tion describes a lazy verification mechanism that attempts to achieve this ideal in the PASIS storage system. It also describes the use of lazy verification to bound the impact of Byzantine-faulty clients (Section 3.8.3) and its application to the garbage collection of unneeded versions (Section 3.8.4).

3.8.1 Lazy Verification Basics

Storage-nodes perform verification using a similar process to a read operation, as described in Section 3.7.2. First, a storage-node finds the latest complete write version. Second, the storage-node performs timestamp validation. If timestamp validation fails, the process is repeated: the storage-node finds the previous complete write version, and performs timestamp validation on that version. Figure 3.6 illustrates the construction and validation of a timestamp in the PASIS read/write protocol. Timestamp validation requires the storage-node to generate a cross checksum based on the erasure-coded fragments it reads. If the generated cross checksum matches that in the timestamp, then the timestamp validates.

Once a block version is successfully verified, a storage-node sets a flag indicating that verification has been performed. (A block version that fails verification is poisonous and is discarded.) This flag is returned to a reading client within the storage-node's response. A client that observes $b + 1$ storage-node responses that indicate a specific block version has been verified need not itself perform timestamp validation, since at least one of the responses must be from a correct storage-node.

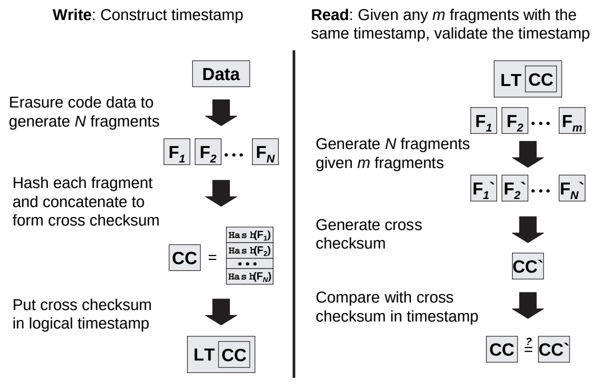

Write: Construct timestamp
Data
Erasure code data to generate *N* fragments
F_1 F_2 … F_N
Hash each fragment and concatenate to form cross checksum
CC = Hash(F_1) Hash(F_2) • • • Hash(F_N)
Put cross checksum in logical timestamp
LT CC
Read: Given any *m* fragments with the same timestamp, validate the timestamp
LT CC
F_1 F_2 … F_m
Generate *N* fragments given *m* fragments
F_1` $F_2$` … F_N`
Generate cross checksum
CC`
Compare with cross checksum in timestamp
CC ≟ CC`

Figure 3.6: Illustration of the construction and validation of timestamps.

When possible, lazy verification is scheduled during idle time periods. Such scheduling minimizes the impact of verification on foreground requests. Significant idle periods exist in most storage systems, due to the bursty nature of storage workloads. Golding et al. [Golding95] evaluated various idle time detectors and found that a simple timer-based idle time detector accurately predicts idle time periods for storage systems. Another study showed that this type of idle time detector performs well even in a heavily loaded storage system [Blackwell95].

Although pre-read verification is the ideal, there is no guarantee that sufficient idle time will exist to lazily verify all writes prior to a read operation on an updated block. Then, there is the question of whether to let the client perform verification on its own (as is done in the original PASIS read/write protocol), or to perform verification *on demand*, prior to returning a read response, so that the client need not. The correct decision depends on the workload and current load. For example, in a mostly-read workload with a light system load, it is beneficial if the storage-nodes perform verification; this will save future clients from having to perform verification. In situations where the workload is mostly-write or the system load is high, then it is more beneficial if the clients perform verification; this increases the overall system throughput.

3.8.2 Cooperative Lazy Verification

Each storage-node can perform verification for itself. But, the overall cost of verification can be reduced if storage-nodes cooperate. As stated above, a client requires only $b + 1$ storage-nodes to perform lazy verification for it to trust the result. Likewise, any storage-node can trust a verification result confirmed by $b + 1$ other storage-nodes. Ideally, then, only $b + 1$ storage-nodes would perform lazy verification. Cooperative lazy verification targets this ideal.

With cooperative lazy verification, once a storage-node verifies a block version, it sends a *notify* message to the other storage-nodes. The notify message is a tuple of ⟨block number, timestamp, status⟩, where "status" indicates the result of the verification. Figures 3.7(a) and 3.7(b) illustrate the messages exchanged without and with cooperative lazy verification, respectively.

We first describe the common-case message sequences for cooperative lazy verification, in concurrency- and fault-free operation. A single storage-node initiates lazy verification by performing verification and sending a notify message to the other storage-nodes. Another b storage-nodes then perform verification and send notify messages to the remaining storage-nodes. The remaining storage-nodes will thus receive $b + 1$ identical notify messages, allowing them to trust the notify messages, since at least one storage-node must be correct. To reduce the number of messages required and to distribute the work of performing verification among storage-nodes, each storage-node is responsible for leading the cooperative lazy verification of a distinct range of block numbers.

If a storage-node needs to verify a block before enough notify messages are received, it will perform verification (and send notify messages), even for blocks that are outside the range it is normally responsible for. In the event of faulty storage-nodes or concurrency, the notification messages may not match. The remaining storage-nodes will then perform verification and send notify messages to the other

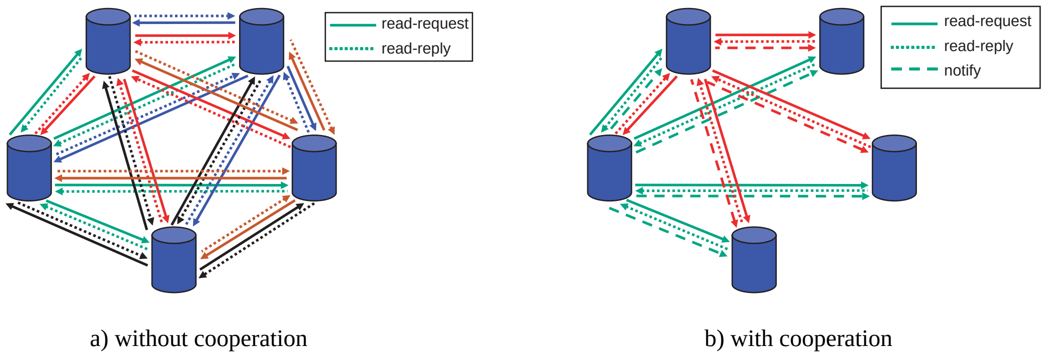


Figure 3.7: Communication pattern of lazy verification (a) without and (b) with cooperation.

storage-nodes, until all storage-nodes have either performed verification or received identical $b + 1$ notify messages. In the worst case, all storage-nodes will perform verification.

The benefit of cooperative lazy verification is a reduction in the number of verification-related messages. Without cooperation, each storage-node must independently perform verification (i.e., effectively perform a read operation). This means each of the N storage-nodes sends read requests to, and receives read responses from, $N - b$ storage-nodes; this yields $O(N^2)$ messages without cooperation. In contrast, the common case for cooperative lazy verification is for just $b + 1$ storage-nodes to perform read requests (to $N - b$ storage-nodes) and then send $N - 1$ notify messages. Even in the worst case in which $2b + 1$ must perform read requests, the communication complexity is still $O(bN)$. For example, even in a minimal system configu ration (N = 4b+1) with $b = 1$, cooperation saves 33% of messages (20 messages with cooperation, versus 30 without), and this benefit improves to over 50% (128 vs. 260) at $b = 3$. Note that storage-nodes that perform cooperative lazy verification need not send notify messages back to storage-nodes from which they received a notify message.

3.8.3 Garbage Collection

If lazy verification passes for a block version, i.e., the block version is complete and its timestamp validates, then block versions with earlier timestamps are no longer needed. The storage-node can delete such obsolete versions. This application of lazy verification is called garbage collection. Garbage collection allows memory and storage capacity at storage-nodes to be reclaimed by deleting unnecessary versions.

In the original presentation of the PASIS read/write protocol, capacity is assumed to be unbounded. In practice, capacity is bounded, and garbage collection is necessary. We distinguish between useful storage, which is the user-visible capacity (i.e., number of blocks multiplied by fragment size), and the history pool. The *history pool* is the capacity used for possible versions. Since the useful storage capacity must usually be determined at system configuration time (e.g., during FORMAT), the maximum size of the history pool will usually be fixed (at raw storage-node capacity minus useful storage capacity). Thus, whenever the history pool's capacity is exhausted, the storage-node must perform garbage collection (i.e., verification plus space reclamation) before accepting new write requests.

Capacity bounds, garbage collection, and liveness. Capacity bounds and garbage collection can impact the liveness semantic of the PASIS read/write protocol. Without capacity bounds, reads and writes are wait-free [Herlihy91]. With a capacity bound on the history pool, however, unbounded numbers of faulty clients could collude to exhaust the history pool with incomplete write operations that cannot be garbage collected—this effectively denies service. This possibility can be eliminated by bounding the number of clients in the system and setting the history pool size appropriately—the history pool size must be the product of the maximum number of clients and the per-client possible version threshold. With this approach, a faulty client can deny service to itself but not to any other client.

Garbage collection itself also affects the liveness semantic, as it can interact with reads in an interesting manner. Specifically, if a read is concurrent to a write, and if garbage collection is concurrent to both, then it is possible for the read not to identify a complete candidate. For example, consider a read concurrent to the write of version-3 with version-2 complete and all other versions at all storage-nodes garbage collected. It is possible for the read to classify version-3 as incomplete, then for the write of version-3 to complete, then for garbage collection to run at all storage-nodes and delete version-2, and finally for the read to look for versions prior to version-3 and find none. Such a read operation must be retried. Because of this interaction, the PASIS read/write protocol with garbage collection is obstruction-free [Herlihy03] (assuming an appropriate history pool size) rather than wait-free.

Performing garbage collection. Garbage collection and lazy verification are tightly intertwined. An attempt to verify a possible version may be induced by history pool space issues, per-client-per-block and per-client thresholds, or the occurrence of a sufficient idle period. Previous versions of a block may be garbage-collected once a later version is verified.

Cooperative lazy verification is applicable to garbage collection. If a storage-node receives notify messages from $b + 1$ storage-nodes, agreeing that verification passed for a given version of a given block, then the storage-node can garbage collect prior versions of that block. Of course, a storage-node may receive $b + 1$ notify messages for a given block that have different timestamp values, if different storage-nodes classify different versions as the latest complete write. In this case, the timestamps are sorted in descending order and the $b + 1^{st}$ timestamp used for garbage collection (i.e., all versions prior to that timestamp may be deleted). This is safe because at least one of the notify messages must be from a correct storage-node.

There is an interesting policy decision regarding garbage collection and repairable candidates. If a client performing a read operation encounters a repairable candidate, it must perform repair and return it as the latest complete write. However, a storage-node performing garbage collection does not have to perform repair. A storage-node can read prior versions until it finds a complete candidate and then garbage collect versions behind the complete candidate. Doing so is safe, because garbage collection "reads" do not need to fit into the linearizable order of operations. By not performing repair, storage-nodes avoid performing unnecessary work when garbage collection is concurrent to a write. On the other hand, performing repair might enable storage-nodes to discard one more block version.

Prioritizing blocks for garbage collection. Careful selection of blocks for verification can improve efficiency, both for garbage collection and lazy verification. Here, we focus on two complementary metrics on which to prioritize this selection process: number of versions and presence in cache.

Since performing garbage collection (lazy verification) involves a number of network messages, it is beneficial to amortize the cost by collecting more than one block version at a time. Each storage-node remembers how many versions it has for each block. By keeping this list sorted, a storage-node can efficiently identify its high-yield blocks: blocks with many versions. Many storage workloads contain blocks that receive many over-writes [Ruemmler03a, Ruemmler03b], which thereby become high-yield blocks. When performing garbage collection, storage-nodes prefer to select *high-yield* blocks. In the common case, all but one of the block's versions will have timestamps less than the latest candidate that passes lazy verification and hence can be deleted (and never verified). Selecting high-yield blocks amortizes the cost of verification over many block versions. This is particularly important when near the per-client possible version threshold or the history pool size, since it minimizes the frequency of on-demand verification.

Block version lifetimes are often short, either because of rapid overwrites or because of create-delete sequences (e.g., temporary files). To maximize the value of each verification operation, storage-nodes delay verification of recently written blocks. Delaying verification is intended to allow rapid sequences to complete, avoiding verification of short-lived blocks and increasing the average version yield by verifying the block once after an entire burst of over-writes. As well, such a delay reduces the likelihood that verification of a block is concurrent with writes to the block. Running verification concurrent to a write, especially if only two local versions exist at a storage-node, may not yield any versions to garbage collect.

Storage-nodes prefer to verify versions while they are in the write-back cache, because accessing them is much more efficient than when going to disk. Moreover, if a block is in one storage-node's cache, it is likely to be in the caches of other storage-nodes, and so verification of that block is likely to not require disk accesses on the other storage-nodes. Garbage collecting versions that are in the write-back cache, before they are sent to the disk, is an even bigger efficiency boost. Doing so eliminates both an initial disk write and all disk-related garbage collection work. Note that this goal matches well with garbage collecting high-yield blocks, if the versions were created in a burst of over-writes.

In-cache garbage collection raises an interesting possibility for storage-node implementation. If only complete writes are sent to disk, which would restrict the history pool size to being less than the cache size, one could use a non-versioning on-disk organization. This is of practical value for implementers who do not have access to an efficient versioning disk system implementation. The results from Section 3.10.3 suggest that this is feasible with a reasonably large storage-node cache (e.g., 500MB or more).

3.9 Implementation

To enable experimentation, we have added lazy verification to the PASIS storage system implementation [Goodson04a]. PASIS consists of storage-nodes and clients. Storage-nodes store fragments and their versions. Clients execute the protocol to read and write blocks. Clients communicate with storage-nodes via a TCP-based RPC interface.

Storage-nodes. Storage-nodes provide interfaces to write a fragment at a logical time, to query the greatest logical time, to read the fragment version with the greatest logical time, and to read the fragment with the greatest logical time before some logical time. With lazy verification, storage-nodes do not return version history or fragments for writes that have been classified as poisonous. In the common case, a client requests the fragment with the greatest logical time. If a client detects a poisonous or incomplete write, it reads the previous version history and earlier fragments.

Each write request creates a new version of the fragment (indexed by its logical timestamp) at the storage-node. The storage-node implementation is based on the S4 object store [Soules03, Strunk00]. A log-structured organization [Rosenblum92] is used to reduce the disk I/O cost of data versioning. Multi-version b-trees [Becker96, Soules03] are used by the storage-nodes to store fragments; all fragment versions are kept in a single b-tree indexed by a 2-tuple $\langle \mathit{blocknumber}, \mathit{timestamp} \rangle$. The storage-node uses a write-back cache for fragment versions, emulating non-volatile RAM.

We extended the base PASIS storage-node to perform lazy verification. Storage-nodes keep track of which blocks have possible versions and perform verification when they need the history pool space, when a possible version threshold is reached, or when idle time is detected. If verification is induced by a per-client-per-block possible version threshold, then that block is chosen for verification. Otherwise, the storage-node's *high-write-count* table is consulted to select the block for which verification is expected to eliminate the highest number of versions. The high-write-count table lists blocks for which the storage-node is *responsible* in descending order of the number of unverified versions associated with each.

Each storage-node is assigned verification responsibility for a subset of blocks in order to realize cooperative lazy verification. Responsibility is assigned by labeling storage-nodes from $0...(N-1)$ and by having only storage-nodes $k \textbf{ mod } N, (k+1) \textbf{ mod } N, \cdots, (k+b) \textbf{ mod } N$ be responsible for block k. In the event of failures, crash or Byzantine, some storage-nodes that are responsible for a particular block may not complete verification. Therefore, any storage-node will perform verification if it does not receive sufficient notify messages before they reach a possible version threshold or exceed history pool capacity. So, while $b + 1$ matching notify messages about a block will allow a storage-node to mark it as verified, failure to receive them will affect only performance.

Client module. The client module provides a block-level interface to higher-level software. The protocol implementation includes a number of performance enhancements that exploit its threshold nature. For example, to improve the responsiveness of write operations, clients return as soon as the minimum number of required success responses is received; the remainder of the requests complete in the background. To improve the read operation's performance, only m read requests fetch the fragment data and version history; the remaining requests only fetch version histories. This makes the read operation more network-efficient. If necessary, after classification, extra fragments are fetched according to the candidate's timestamp.

PASIS supports both replication and an m-of-N erasure coding scheme. If $m = 1$, then replication is employed. Otherwise, our base erasure code implementation stripes the block across the first m fragments; each stripe-fragment is 1 m the length of the original block. Thus, concatenation of the first m fragments produces the original block. (Because "decoding" with the m stripe-fragments is computationally less expensive, the implementation always tries to read from the first m storage-node for any block.) The stripe-fragments are used to generate the $N - m$ code-fragments, via polynomial interpolation within a Galois Field. The implementation of polynomial interpolation is based on publicly available code [Dai-b] for information dispersal [Rabin89]. As detailed in Section 3.5, this code was modified to make use of stripe-fragments and to add an implementation of Galois Fields of size 2^8 that use lookup tables for multiplication. MD5 is used for all hashes; thus, each cross checksum is $N \times 16$ bytes long.

We extended the PASIS client module to use the flag described in Section 3.8.1 to avoid the verification step normally involved in every read operation, when possible. It checks the "verified" flag returned from each contacted storage-node and does its own verification only if fewer than $b + 1$ of these flags are set. In the normal case, these flags will be set and read-time verification can be skipped.

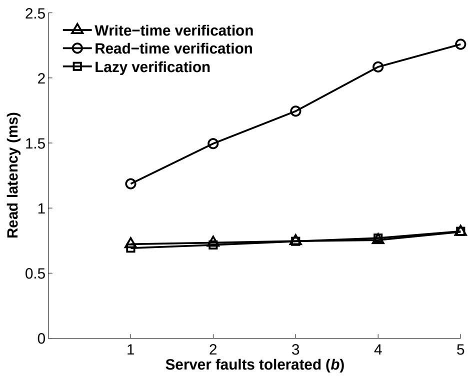
3.10 Evaluation

This section quantifies benefits of lazy verification. It shows that significant increases in write throughput can be realized with reasonably small history pool sizes. In fact, lazy verification approaches the ideal of zero performance cost for verification. It also confirms that the degradation-of-service vulnerability inherent to delayed verification can be bounded with minimal performance impact on correct clients.

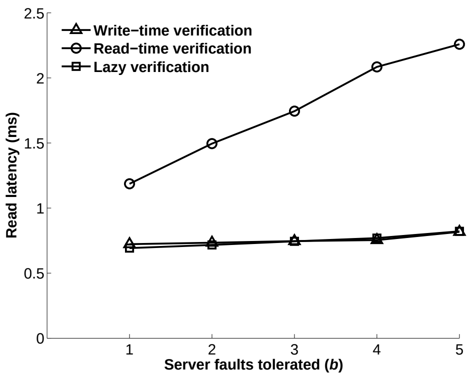
3.10.1 Experimental Setup

All experiments are performed on a collection of Intel Pentium 4 2.80GHz computers, each with 1GB of memory and an Intel PRO/1000 NIC. The computers are connected via an HP ProCurve Switch 4140gl specified to perform 18.3Gbps/35.7mpps. The computers run Linux kernel 2.6.11.5 (Debian 1:3.3.4-3).

Micro-benchmark experiments are used to focus on performance characteristics of lazy verification. Each experiment consists of some number of clients performing operations on a set of blocks. Experiments are run for 40 seconds and measurements are taken after 20 seconds. (The 20 second warm-up period is sufficient to ensure steady-state system operation.) For the benchmarks used in this section, the working set and history pool are sized to fit in the storage-nodes' RAM caches. Given that the storage-node cache is assumed to be non-volatile, this eliminates disk activity and allows focus to stay on the computation and networking costs of the protocol.



a) read latency



b) write latency

Figure 3.8: Operation latencies for different verification policies.

3.10.2 Verification Policies and Operation Latencies

A storage system's verification policy affects client read and write operation latencies. We ran an experiment to measure the operation latencies for three different verification policies: proactive write-time verification, read-time verification, and lazy verification. The write-time verification policy is emulated by having each storage-node perform verification for a block immediately after it accepts a write request—this is similar to the system described by Cachin and Tessaro [Cachin05b]. With the read-time verification policy, the client incurs the cost of verification on every read operation.

We ran experiments for different numbers of tolerated server faults (from b = 1 to b = 5). For each experiment, we used the minimal system configuration: $N = 4b + 1$. The erasure coding reconstruction threshold m equals $b + 1$ in each experiment. This provides the maximal space- and network-efficiency for the value of N employed. A single client performs one operation at a time. The client workload is an equal mix of read and write operations. After each operation, the client sleeps for 10ms. This introduces idle time into the workload.

Figure 3.8(a) shows the client write latency for the different verification policies. The write-time verification policy has a higher write latency then the two other verification policies. This is because the storage-nodes perform verification in the critical path of the write operation. For both the read-time and lazy verification policies, the client write latency increases slightly as more faults are tolerated. This is due to the increased computation and network cost of generating and sending fragments to more servers.

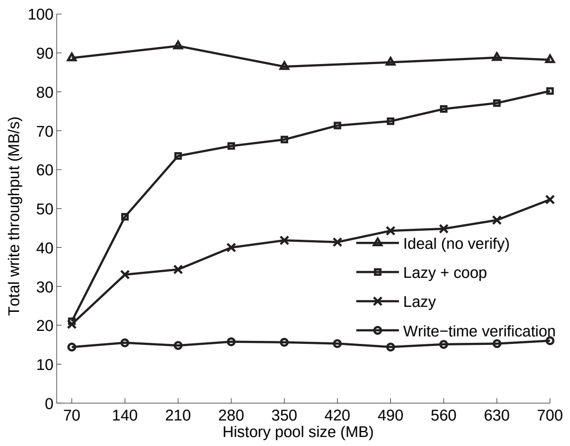


Figure 3.9: Write throughput, as a function of history pool size, for four ver

Figure 3.8(b) shows the client read latency for the different verification policies. For the read-time verification policy, the client read latency increases as more faults are tolerated. This is because the computation cost of generating the fragments to check the cross checksum and validate the timestamp increases as b increases. For the write-time verification policy, verification is done at write-time, and so does not affect read operation latency. For the lazy verification policy, sufficient idle time exists in the workload for servers to perform verification in the background. As such, the read latency for the lazy verification policy follows that for the write-time verification policy.

These experiments show that, in workloads with sufficient idle time, lazy verification removes the cost of verification from the critical path of client read and write operations. Lazy verification achieves the fast write operation latencies associated with read-time verification, yet avoids the overhead of client verification for read operations.

3.10.3 Impact on Foreground Requests

In many storage environments, bursty workloads will provide plenty of idle time to allow lazy verification and garbage collection to occur with no impact on client operation performance. To explore bounds on the benefits of lazy verification, this section evaluates lazy verification during non-stop high-load with no idle time. When there is no idle time, and clients do not exceed the per-client or per-block-per-client thresholds, verification is induced by history pool exhaustion. The more efficient verification and garbage collection are, the less impact there will be on client operations.

In this experiment, four clients perform write operations on 4096 32KB blocks, each keeping eight operations in progress at a time and randomly selecting a block for each operation. Also, the system is configured to use $N = 5$ storage-nodes, while tolerating one Byzantine storage-node fault ($b = 1$) and employing 2-of-5 erasure-coding (so, each fragment is 16KB in size, and the storage-node working set is 64MB).

Figure 3.9 shows the total client write throughput, as a function of history pool size, with different verification policies. The top and bottom lines correspond to the performance ideal (zero-cost verification) and the conventional approach of performing verification during each write operation, respectively. The ideal of zero-cost verification is emulated by having each storage-node replace the old version with the new without performing any verification at all. As expected, neither of these lines is affected by the history pool size, because versions are very short-lived for these schemes. Clearly, there is a significant performance gap (5×) between the conventional write-time verification approach and the ideal.

The middle two lines correspond to use of lazy verification with (“Lazy + coop”) and without (“Lazy”) cooperative lazy verification. Three points are worth noting. First, with lazy verification, client write throughput grows as the history pool size grows. This occurs because a larger history pool allows more versions of each block to accumulate before history pool space is exhausted. As a result, each verification can be amortized over a larger number of client writes. (Recall that all earlier versions can be garbage collected once a later version is verified.) Second, with a reasonably-sized 700MB history pool size, cooperative lazy verification provides client write throughput within 9% of the ideal. Thus, even without idle time, lazy verification eliminates most of the performance costs of verification and garbage collection, providing a factor of four performance increase over conventional write-time verification schemes. Third, cooperative lazy verification significantly reduces the impact of verification, increasing client write throughput by 53–86% over non-cooperative lazy verification for history pool sizes over 200MB.

3.11 Summary

Lazy verification can significantly improve the performance of Byzantine fault-tolerant distributed storage systems that employ erasure-coding. It shifts the work of verification out of the critical path of client operations and allows significant amortization of work. Measurements show that, for workloads with idle periods, the cost of verification can be hidden from both the client read and write operation. In workloads without idle periods, lazy verification and its concomitant techniques—storage-node cooperation and prioritizing the verification of high-yield blocks—provides a factor of four greater write bandwidth than a conventional write-time verification strategy.

4 FAULT SCALABLE BYZANTINE FAULT-TOLERANT SERVICES: QUERY/UPDATE PROTOCOL

4.1 Introduction

Today's distributed services face unpredictable network delays, arbitrary failures induced by increasing software complexity, and even malicious behavior and attacks from within compromised components. Like many, we believe it is important to design for these eventualities. In fact, it is increasingly important to design for multiple faults and for graceful degradation in the face of faults. Services are being designed for a massive scale in open clusters (e.g., like the Farsite project [Adya02]), over the WAN (e.g., like the OceanStore project [Rhea03]), and in administration-free clusters that allow components to fail in place (e.g., like the Collection of Intelligent Bricks project [Morris02]). In such settings, timing faults (such as those due to network delays and transient network partitions) and failures of multiple components, some of which may be Byzantine in nature (i.e., arbitrary or malicious), may arise. Moreover, rates of server and client crashes in such settings will likely be higher than in carefully-administered settings.

There is a compelling need for services (e.g., namespace, key management, metadata, and LDAP) and distributed data structures (e.g., b-trees, queues, and logs) to be efficient and fault-tolerant. Various services and data structures that efficiently tolerate benign failures have been developed (e.g., Paxos [Lamport98], Boxwood [MacCormick04], and Chain Replication [VanRenesse04]). In the context of tolerating Byzantine faults, the focus has been on building services via replicated state machines using an agreement-based approach [Lamport78, Schneider90]. Examples of such systems include the BFT system of Castro and Liskov [Castro02] and the SINTRA system of Cachin and Poritz [Cachin02].

A *fault-scalable* service is defined to be one in which performance degrades gradually, if at all, as more server faults are tolerated. Our experience is that Byzantine fault-tolerant agreement-based approaches are not fault-scalable: their performance drops rapidly as more faults are tolerated because of server-to-server broadcast communication and the requirement that all correct servers process every request.

This chapter describes the Query/Update (Q/U) protocol as a fault-scalable alternative to these approaches. The Q/U protocol is an efficient, optimistic, quorum-based protocol for building Byzantine fault-tolerant services. Services built with the Q/U protocol are fault-scalable. The Q/U protocol provides an *operations-based interface* that allows services to be built in a similar manner to replicated state machines [Lamport78, Schneider90]. Q/U objects export interfaces comprised of deterministic methods: *queries* that do not modify objects and *updates* that do. Like a replicated state machine, this allows objects to export narrow interfaces that limit how faulty clients may modify objects [Schneider90]. A client's update of an object is conditioned on the object version last queried by the client. An update succeeds only if the object has not been modified since the client's last query in which case it is retried. Moreover, the Q/U protocol supports *multi-object updates* that atomically update a set of objects conditioned on their respective object versions.

The Q/U protocol operates correctly in an asynchronous model (i.e., no timing assumptions are necessary for safety), tolerates Byzantine faulty clients, and tolerates Byzantine faulty servers. Queries and updates in the Q/U protocol are strictly serializable [Bernstein87]. In a benign execution (i.e., an execution in which components act according to specification or crash), queries and updates are obstruction-free [Herlihy03]. The "cost" of achieving these properties with the Q/U protocol, relative to other approaches, is an increase in the number of required servers: the Q/U protocol requires $5b+1$ servers to tolerate b Byzantine faulty servers, whereas most agreement-based approaches require $3b+1$ servers. Given the observed performance of the Q/U protocol, we view this as a good trade-off—the cost of servers continues to decline, while the cost of service failures does not.

The Q/U protocol achieves its performance and faults-scalability through novel integration of a number of techniques. Optimism is enabled by the use of non-destructive updates at versioning servers, which

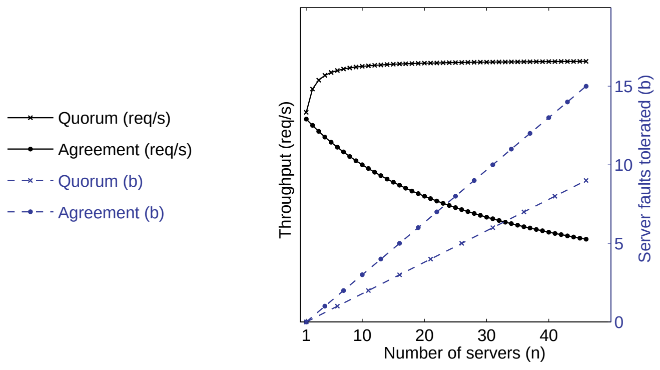


Figure 4.1: Illustration of fault-scalability. As the number of server faults tolerated increases (right axis, dashed lines), the number of servers required (x-axis) by quorum- and agreement-based protocols increases. In theory, throughput (left axis, solid lines) of quorum-based approaches (e.g., based on a threshold quorum) is sustained as more faults are tolerated, whereas agreement-based approaches do not have this property. In practice, throughput of prototype Q/U-based services degrades somewhat as more faults are tolerated.

permits operations to efficiently resolve contention and/or failed updates, e.g., by querying earlier object versions and completing partial updates. We leverage this versioning together with a logical time-stamping scheme in which each update operation is assigned a timestamp that depends both on the contents of the update and the object state on which it is conditioned. It is thus impossible for a faulty client to submit different updates at the same timestamp—the updates intrinsically have different timestamps—and so, reaching agreement on the update at a given timestamp is unnecessary. We combine these techniques with quorums and a strategy for accessing them using a *preferred quorum* per object so as to make server-to-server communication an exceptional case. Finally, we employ efficient cryptographic techniques. Our integration of these techniques has enabled, to our knowledge, the first fault-scalable, Byzantine-resilient implementation for arbitrary services.

We implemented a prototype library for the Q/U protocol. We used this library to build two services: a metadata service that exports NFSv3 metadata methods, and a counter object that exports increment (**increment**) and fetch (**fetch**) methods. Measurements of these prototype services support our claims: the prototype Q/U services are efficient and fault-scalable. In contrast, the throughput of BFT [Castro02], a popular agreement-based Byzantine fault-tolerant replicated state machine implementation, drops sharply as the number of faults tolerated increases. The prototype Q/U-based counter outperforms a similar counter object implemented with BFT at all system sizes in contention-free experiments. More importantly, it is more fault-scalable. Whereas the performance of the Q/U-based counter object decreases by 36% as the number of faults tolerated is increased from one to five, the performance of the BFT-based counter object decreases by 83%.

4.2 Efficiency and Scalability

In a Byzantine fault-tolerant quorum-based protocol (e.g., [Malkhi98a, Malkhi00a, Chockler01, Martin02, Zhou02, Fry04]), only quorums (subsets) of servers process each request, and server-to-server communication is generally avoided. In a Byzantine fault-tolerant agreement-based protocol (e.g., [Bracha85, Reiter95, Kihlstrom01, Castro02, Cachin02, Kursawe02]), on the other hand, every server processes each request and performs server-to-server broadcast. As such, these approaches exhibit fundamentally different fault-scalability characteristics.

The expected fault-scalability of quorum- and agreement-based approaches is illustrated in Figure 4.1. A protocol that is fault-scalable provides throughput that degrades gradually, if at all, as more server faults are tolerated. Because each server must process every request in an agreement-based protocol, increasing the number of servers cannot increase throughput. Indeed, since server-to-server broadcast is required, the useful work each server can do decreases as the number of servers increases. As illustrated, though, agreement-based protocols generally require fewer servers than quorum-based protocols for a given degree of fault-tolerance.

With quorum-based protocols, as the number of server faults tolerated increases, so does the quorum size. As such, the work required of a client grows as quorum size increases. Servers do a similar amount of work per operation regardless of quorum size. However, the Q/U protocol does rely on some cryptographic techniques (i.e., authenticators which are discussed in x3) whose costs grow with quorum size.

4.2.1 Efficiency

Much of the efficiency of the Q/U protocol is due to its optimism. During failure- and concurrency-free periods, queries and updates occur in a single phase. To achieve *failure atomicity*, most pessimistic protocols employ at least two phases (e.g., a prepare and a *commit* phase). The optimistic approach to failure atomicity does not require a prepare phase; however, it does introduce the need for clients to repair (write-back) inconsistent objects. To achieve *concurrency atomicity*, most pessimistic protocols either rely on a central point of serialization (e.g., a primary) or employ locks (or leases) to suppress other updates to an object while the lock-holder queries and updates the object. The optimistic approach to concurrency atomicity [Kung81] does not require lock acquisition, but does introduce the possibility that updates are rejected (and that clients may livelock).

The Q/U protocol relies on versioning servers for its optimism. Every update method that a versioning server invokes results in a new object version at that server. Queries complete in a single phase so long as all of the servers in the quorum contacted by a client share a common latest object version. Updates complete in a single phase so long as none of the servers in the quorum contacted by a client have updated the object since it was queried by the client. To promote such optimistic execution, clients introduce locality to their quorum accesses and cache object version information. Clients initially send requests to an object's *preferred quorum* and do not issue queries prior to updates for objects whose version information they cache. Finally, versioning servers reduce the cost of protecting against Byzantine faulty clients, since servers need not agree on client requests before processing them.

Many prior protocols, both pessimistic and optimistic, make use of versions and/or logical timestamps. One difference with prior protocols is that server retention of object versions is used to efficiently tolerate Byzantine faulty clients. Most protocols that tolerate Byzantine faulty clients rely on digital signatures or server-to-server broadcasts. Another difference is that there is no concept of a commit phase in the Q/U protocol (not even a lazy commit).

4.2.2 Throughput Scalability

The primary benefit that the Q/U protocol gains from the quorum-based approach is fault-scalability. Quorum-based protocols can also exhibit throughput-scalability: additional servers, beyond those necessary for providing the desired fault-tolerance, can increase throughput [Naor98, Malkhi00b]. The experience of database practitioners suggests that it may be difficult to take advantage of quorum-based *throughput-scalability* though. For example, Jiménez-Peris et al. recently concluded that a write-all read-one approach is better for a large range of database applications than a quorum-based approach [Jimenez-Peris03]. But their analysis ignores concurrency control and is based on two phase commit-based data replication with fail-stop failures in a synchronous model. The Q/U protocol provides both concurrency and failure atomicity, provides service replication rather than data replication, relies on no synchrony assumptions, and relies on few failure assumptions. As another example, Gray et al. identify that the use of quorum-based data replication in databases leads to the *scaleup pitfall*: the higher the degree of replication, the higher the rate of deadlocks or reconciliations [Gray96]. But, databases are not designed to scale up gracefully. The ability to update multiple objects atomically with the Q/U protocol allows services to be decomposed into *fine-grained* Q/U objects. Fine-grained objects reduce per-object contention, making optimistic execution more likely, enabling parallel execution of updates to distinct objects, and improving overall service throughput. If a service can be decomposed into fine-grained Q/U objects such that the majority of queries and updates are to individual objects, and the majority of multi-object updates span a small number of objects, then the scaleup pitfall can be avoided. Our experience building a Q/U-NFSv3 metadata service suggests that it is possible to build a substantial service comprised of fine-grained objects. For example, most metadata operations access a single object or two distinct objects. (The only operation that accesses more than two objects is the RENAME operation which accesses up to four objects.) In response to some criticisms of quorum-based approaches, Wool argues that quorum-based approaches are well-suited to large scale distributed systems that tolerate malicious faults [Wool98]. Wool's arguments support our rationale for using a quorum-based approach to build fault-scalable, Byzantine fault-tolerant services.

Many recent fault-tolerant systems achieve throughput-scalability by partitioning the services and data structures they provide (e.g., [Litwin00, Gribble00, Adya02, Rhea03, MacCormick04, VanRenesse04]). By partitioning different objects into different server groups, throughput scales with the addition of servers. However, to partition, these systems either forego the ability to perform operations that span objects (e.g., [Litwin00, Gribble00, Rhea03, VanRenesse04]) or make use of a special protocol/service for "transactions" that span objects (e.g., [Adya02, MacCormick04]). Ensuring the correctness and atomicity of operations that span partitions is complex and potentially quite expensive, especially in an asynchronous, Byzantine fault-tolerant manner. To clarify the problem with partitioning, consider a RENAME operation that moves files between directories in a metadata service. If different server groups are responsible for different directories, an interserver group protocol is needed to atomically perform the RENAME operation (e.g., as is done in Farsite [Adya02]). Partitioning a service across server groups introduces dependencies among server groups. Such dependencies necessarily reduce the reliability of the service, since many distinct server groups must be available simultaneously.

4.3 The Query/Update Protocol

This section begins with a discussion of the system model and an overview of the Q/U protocol for individual objects. We discuss constraints on the quorum system needed to ensure the correctness of the Q/U protocol. We present detailed pseudo-code and discuss implementation details and design trade-offs. To make the Q/U protocol's operation more concrete, we describe an example system execution. We describe the extension of the Q/U protocol for individual objects to multiple objects. Finally, we discuss the correctness of the Q/U protocol.

4.3.1 System Model

To ensure correctness under the broadest possible conditions, we make few assumptions about the operating environment. An asynchronous timing model is used; no assumptions are made about the duration of message transmission delays or the execution rates of clients and servers (except that they are non-zero and finite). Clients and servers may be Byzantine faulty [Malkhi01]: they may exhibit arbitrary, potentially malicious, behavior. Clients and servers are assumed to be computationally bounded so that cryptographic primitives are effective. Servers have persistent storage that is durable through a crash and subsequent recovery.

The server failure model is a hybrid failure model [Thambidurai88] that combines Byzantine failures with crash-recovery failures (as defined by Aguilera et al. [Aguilera00]). A server is *benign* if it is correct or if it follows its specification except for crashing and (potentially) recovering; otherwise, the server is *malevolent*. Since the Byzantine failure model is a strict generalization of the crash-recovery failure model, another term—malevolent—is used to categorize those servers that in fact exhibit out-of-specification, non-crash behavior. A server is *faulty* if it crashes and does not recover, crashes and recovers *ad infinitum*, or is malevolent.

We extend the definition of a fail prone system given in Malkhi and Reiter [Malkhi98a] to accommodate this hybrid failure model. We assume a universe U of servers such that $|U| = n$. The system is characterized by two sets: $\mathcal{T} \subseteq 2^U$ and $\mathrm{B} \subseteq 2^U$ (the notation 2^{set} denotes the power set of set). In any execution, all faulty servers are included in some $T \in \mathcal{T}$ and all malevolent servers are included in some $\mathrm{B} \in \mathcal{B}$. It follows from the definitions of faulty and malevolent that $\mathrm{B} \subseteq \mathrm{T}$.

The Q/U protocol is a quorum-based protocol. A quorum system $\mathcal{Q} \subseteq 2^U$ is a non-empty set of subsets of U, every pair of which intersect; each $Q \in \mathcal{Q}$ is called a quorum. Constraints on the quorum system are described in Section 4.3.3. For simplicity, in the pseudo-code (Section 4.3.4) and the example execution (Section 4.3.6) presented here, we focus on threshold quorums. With a threshold quorum system, a fail prone system can simply be described by bounds on the total number of faulty servers: there are no more than t faulty servers of which no more than $b \leq t$ are malevolent. Point-to-point authenticated channels exist among all of the servers and between all of the clients and servers. An infrastructure for deploying shared symmetric keys among pairs of servers is assumed. Finally, channels are assumed to be unreliable, with the same properties as those used by Aguilera et al. in the crash-recovery model (i.e., channels do not create messages, channels may duplicate messages a finite number of times, and channels may drop messages a finite number of times) [Aguilera00]. Such channels can be made reliable by repeated resends of requests.

4.3.2 Overview

This section overviews the Q/U protocol for an individual object.

Terminology. Q/U objects are replicated at each of the n servers in the system. Q/U objects expose an operations-based interface of deterministic methods. Read-only methods are called *queries*, and methods that modify an object's state are called *updates*. Methods exported by Q/U objects take *arguments* and return *answers*. The words "operation" and "request" are used as follows: clients *perform operations* on an object by issuing *requests* to a *quorum* (subset) of servers. A server *receives* requests; if it *accepts* a request, it *invokes* a *method* on its local object replica.

Each time a server invokes an update method on its copy of the object, a new *object version* results. The server retains the object version as well as its associated *logical timestamp* in a version history called the *replica history*. Servers return replica histories to clients in response to requests.

A client stores replica histories returned by servers in its *object history set* (OHS), which is an array of replica histories indexed by server. In an asynchronous system, clients only receive responses from a subset of servers. As such, the OHS represents the client's partial observation of global system state at some point in time. Timestamps in the replica histories in the OHS are referred to as *candidates*. Candidates are *classified* to determine which object version a client method should be invoked on at the servers. Note that there is a corresponding object version for every candidate. Figure 4.2 illustrates candidates, replica histories, and the object history set.

Client side. Because of the optimistic nature of the Q/U protocol, some operations require more client steps to complete than others. In an optimistic execution, a client completes queries and updates in a single phase of communication with servers. In a non-optimistic execution, clients perform additional repair phases that deal with failures and contention.

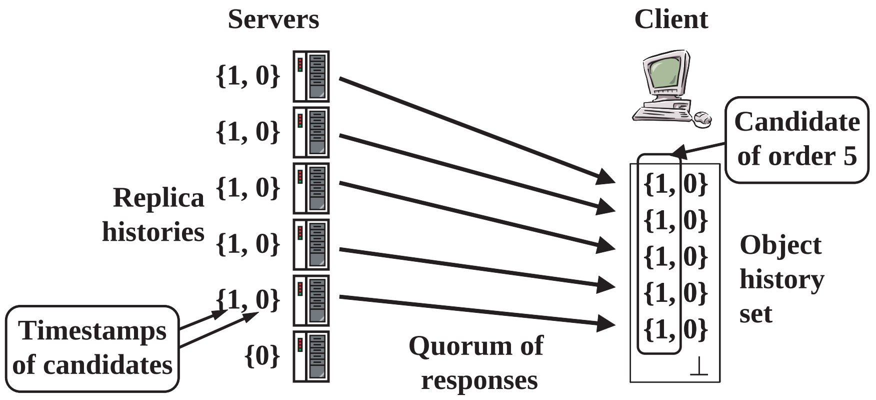


Figure 4.2: Example of client and server state.

The operations-based interface allows clients to send operations and receive answers from the service; this provides light-weight client-server communication relative to reading and writing entire objects. To perform an operation, a client first retrieves the object history set. A client’s operation is said to *condition on* its object history set (the *conditioned-on OHS*). The client places the object history set, the method to invoke, and the arguments for the method in a request it sends to servers. By sending the object history set, the client communicates information to the servers about global system state.

Both clients and servers classify the conditioned-on OHS to determine which object version a client operation should be performed on at the servers. Classification of a candidate is based on the subset of server replica histories in the object history set in which it appears. (For threshold quorums, classification of a candidate is based on its *order*, the number of replica histories in the object history set in which it appears.) If all of the replica histories in the conditioned-on OHS have the same latest candidate, this candidate is classified as complete. This is illustrated in Figure 4.2.

The optimistic nature of the Q/U protocol allows a client to complete queries, and updates conditioned on a cached object history set, in a single phase of client-server communication during failure- and concurrency-free access. Clients cache object history sets. This allows clients to avoid retrieving an object history set before every update operation. So long as no other clients update the object, the cached object history set remains current. Only the quorum of servers that accepted the last update operation has the latest object version. As such, it is most efficient for these servers to process the subsequent update operation. To promote locality of access, and thus efficiency, each object has a *preferred quorum* at which clients try to access it first. In the case of server failures, clients may have to access a non-preferred quorum. Single phase operations are premised on the client's cached object history set being current (concurrency-free access) and on accessing an object via its preferred quorum (failure-free access).

Server side. Upon receipt of a request, a server first validates the integrity of the conditioned-on OHS. Each server pairs an *authenticator* with its replica history and clients include the authenticators in the conditioned-on OHS. Authenticators are lists of HMACs that prevent malevolent clients and servers from fabricating replica histories for benign servers. Servers cull replica histories from the conditioned-on OHS when they cannot validate the authenticator. Authenticators are needed because servers do not directly exchange replica histories with one another; they are communicated between servers via the client.

If the conditioned-on OHS passes integrity validation, the server performs classification. Next, a server validates that the conditioned-on OHS is current. The server does this by comparing the timestamps of candidates in its replica history with the current timestamp classification identified for the conditioned-on OHS. If the server has a higher timestamp in its replica history then the current timestamp identified by classification, the conditioned-on OHS is not current. Validating currentness ensures that servers invoke methods on the latest complete object version, rather than one of its predecessors. If all validation passes, a server accepts the request and invokes the requested method. The method is invoked on the object version corresponding to the latest candidate in the conditioned-on OHS. If the method is an update, a new version of the object results. The timestamp for the resulting object version is a deterministic function of the conditioned-on OHS and the operation performed. As such, all servers that receive the same operation and object history set create the same object version and construct the same timestamp for that object version. Timestamps are constructed so that they always increase in value. The server updates its replica history with the resulting timestamp and stores the resulting object version indexed by that timestamp. The server sends a response to the client, indicating success, including the answer to the method, its replica history, and its authenticator. The client updates its object history set with the returned replica histories and authenticators. Once the client receives a quorum of responses that return success, the operation returns successfully.

Tolerating failures. If a server crashes, some quorums may become unavailable. This may lead to clients *probing* additional servers to collect a quorum of responses (probing is the term used to describe finding a live quorum). To protect against malevolent components, timestamps contain the client ID, the operation (method and arguments), and the conditioned-on OHS. Because the operation is tied to the timestamp, an operation can only complete successfully if a client sends the same operation to all servers in a quorum. This mechanism makes it impossible for malevolent clients to force object versions at different benign servers with the same timestamp to differ. The authenticators in the object history set make it impossible for malevolent components to forge replica histories of benign servers; in conjunction with classification rules (see Section 4.3.3), this ensures that benign servers only accept requests conditioned on the latest complete object version.

Concurrency and repair. Only one update of a specific object version can complete successfully. As such, concurrent accesses of an object may fail and result in replica histories at different servers that have a different latest candidate. In such a situation, a client must bring the servers into a consistent state. To do so, a client performs repair.

Repair consists of a sequence of two special operations: first a *barrier* is performed, and then a *copy* is performed. Barrier candidates have no data associated with them and so are safe for servers to accept during periods of contention. Indeed, if contention is indicated by the conditioned-on OHS sent to a server, the server can only accept a barrier candidate. Barrier operations allow clients to safely advance logical time in the face of contention; barriers prevent operations with earlier timestamps from completing. Once sufficient servers have accepted barriers, all contending operations have been suppressed. At this point the client can perform a *copy* operation. The copy operation copies the latest object version prior to the barrier forward in logical time. Such an object version is either the result of one of the contending operations that triggered repair or the object version these operations conditioned on. In the former case, none of the contending operations was successful, and in the latter case, one of the contending operations was successful. Once a copy operation successfully completes, then it is possible for another update or query to be performed.

Classification of the OHS dictates whether or not repair is necessary and, if so, whether a barrier or copy must be performed. Since both clients and servers base their actions on the OHS, both perform the action dictated by classification.

In the face of contention, clients may have to repeatedly perform barrier and copy operations: different clients could be copying different repairable candidates forward, never completing the copy operation. Client backoff is relied upon to promote progress when there is such contention. Since servers always return their latest replica history to the client, the client can, after updating its object history set, observe the effect of its operation.

Non-preferred quorum access. To be efficient, clients normally access an object via the object's preferred quorum. If a client accesses a non-preferred quorum, some servers that receive requests may not have the conditioned-on object version. In such a scenario, the server must *object sync* to retrieve the conditioned-on object version. The *sync-server* requests the needed object data from *host-servers*. The replica histories in the conditioned-on OHS provide the sync server with information about which other servers are host-servers. Responses from at least $b + 1$ host-servers are required for the sync-server to validate that the data it receives is correct.

4.3.3 Classification and Constraints

In an asynchronous system with failures, clients can only wait for a subset of servers to reply. As such, object history sets are a partial observation of the global system state. Based on these partial observations, the latest candidate is classified as *complete*, *repairable*, or *incomplete*. With perfect global information, it would be possible to directly observe whether or not the latest candidate is *established*. An established candidate is one that is accepted at all of the benign servers in some quorum. Constraints on the quorum system, in conjunction with *classification rules*, ensure that established candidates are classified as repairable or complete. In turn, this ensures that updates are invoked on the latest object version.

Repairing a candidate, i.e., performing a barrier and then copying the candidate, is a fundamental aspect of the Q/U protocol. To allow us to state the classification rules and quorum intersection properties regarding repair in the Q/U protocol, we define repairable sets. Each quorum $Q \in \mathcal{Q}$ defines a set of repairable sets $\mathcal{R}(Q) \subseteq 2^Q$. Given a set of server responses S that share the same candidate, the classification rules for that candidate are as follows:

$$\text{classify}(S) = \begin{cases} \textit{complete} & \text{if } \exists Q \in \mathcal{Q} : Q \subseteq S \\ \textit{repairable} & \text{if } (\forall Q \in \mathcal{Q} : Q \not\subseteq S) \wedge \\ & (\exists Q \in \mathcal{Q}\,, R \in \mathcal{R}(Q): \\ & R \subseteq S), \\ \textit{incomplete} & \text{otherwise.} \end{cases} \qquad (4.1)$$

Constraints on the quorum system that we require are as follows:

$$\forall T\ \exists Q : Q \cap T = \emptyset; \qquad (4.2)$$

$$\forall Q_i, Q_j \in \mathcal{Q}; \forall B \in \mathcal{B}; \exists R \in \mathcal{R}(Q_i) : R \subseteq Q_i \cap Q_j \setminus B; \qquad (4.3)$$

$$\forall Q_i, Q_j \in \mathcal{Q}, \forall B \in \mathcal{B}, \forall R \in \mathcal{R}(Q_j) : Q_i \cap R \not\subset B. \qquad (4.4)$$

Constraint (4.2) ensures that operations at a quorum may complete in an asynchronous system. Constraint (4.3) ensures that some repairable set of an established candidate is contained in every other quorum. This constraint ensures that an established candidate is always classified as repairable or complete. For example, if a candidate is established at quorum Q_i, then a subsequent quorum access to Q_j is guaranteed to observe a repairable set R of Q_i despite any malevolent servers.

A candidate that, in some execution, could be classified as repairable is a potential candidate. Specifically, a potential candidate is one that is accepted at all of the benign servers in some repairable set. Constraint (4.4) ensures that an established candidate intersects a potential candidate at at least one benign server. This constraint ensures that at most one of any concurrent updates in the Q/U protocol establishes a candidate and that, if a candidate is established, no other concurrent updates yield a potential candidate. The logic that dictates which type of operation (method, barrier, or copy) must be performed is intimately connected with the classification rules and with the quorum system constraints. The constraints allow for multiple potential candidates. As such, there could be multiple distinct potential candidates conditioned on the same object version. However, if there are any potential candidates, constraint (4) precludes there from being any established candidates. Since none of these potential candidates can ever be established, the Q/U protocol requires barriers to safely make progress. Since barriers do not modify object state, they can be accepted at servers if the latest candidate is classified as incomplete or repairable. Once a barrier is established, it is safe to copy the latest object version forward (whether it is a potential candidate or an established candidate). And, if the copy establishes a candidate, then the corresponding object version can be conditioned on.

Classification and threshold quorums. Many quorum constructions can be used in conjunction with the Q/U protocol (e.g., those in [Malkhi98a, Malkhi00a]). We have implemented threshold (majority) quorum constructions and recursive threshold quorum constructions from [Malkhi00a]. However, since the focus of this portion of our research is on fault-scalability, we focus on the smallest possible threshold quorum construction for a given server fault model.

For the remainder of this section, we consider a threshold quorum system in which all quorums $Q \in \mathcal{Q}$ are of size q, all repairable sets $R \in \mathcal{R}(Q)$ are of size r, all faulty server sets $T \in \mathcal{T}$ are of size t, all malevolent server sets $B \in \mathcal{B}$ are of size b, and the universe of servers is of size n. In such a system, (4.2) implies that $q + t \le n$, (4.3) implies that $2q - b - n \ge r$, and (4.4) implies that $q+r - n > b$. Rearranging these inequalities allows us to identify the threshold quorum system of minimal size given some t and b:

$n = 3t + 2b + 1;$

$q = 2t + 2b + 1;$

$r = t + b + 1.$

As such, if $b = t$, then $n = 5b + 1$, $q = 4b + 1$, and $r = 2b + 1$. For example, if $b = 1$, then $n = 6$, $q = 5$, and $r = 3$. If $b = 4$, then $n = 21$, $q = 17$, and $r = 9$. For such a threshold quorum system, classification is based on the order of the candidate. The order is simply the number of servers that reply with the candidate. (This is illustrated in Figure 4.2.) The classification rules (1) for threshold quorums are as follows:

$$\text{classify}(Order) = \begin{cases} complete & \text{if } q \le Order, \\ repairable & \text{if } r \le Order < q, \\ incomplete & \text{otherwise.} \end{cases} \qquad (4.5)$$

4.3.4 Q/U Protocol Pseudocode

Figure 4.3 shows the pseudo-code for the Q/U protocol for a single object. To simplify the pseudo-code, it is written especially for threshold quorums. The symbols used are summarized in the caption. Structures, enumerations and types used in the pseudo-code are given on lines 100-107. The definition of a *Candidate* includes the conditioned-on timestamp LT_{CO}, even though it can be generated from *LT.OHS*. It is included because it clearly indicates the conditioned-on object version. The equality (=) and less than (<) operators are well-defined for timestamps; less than is based on comparing *Time*, *BarrierFlag* (with FALSE < TRUE), *ClientID*, *Operation* (lexigraphic comparison), and then *OHS* (lexigraphic comparison).

Clients initialize their object history set to a well-known value (line 200). Two example methods for a simple counter object are given: a query **c_fetch** and an update **c_increment**. The implementation of these two functions are similar, the only difference being the definition of the operation in each function (lines 300 and 400). First, requests based on the operation are issued to a quorum of servers (line 301). The OHS passed to **c_quorum_rpc** is the clients cached OHS. If this OHS is not current, servers reject the operation and return their latest replica history, in an effort to make the client's OHS current. If the operation is accepted at the quorum of servers (line 302), the function returns. Otherwise, the client goes into a repair and retry cycle until the operation is accepted at a quorum of servers (lines 302-306).

The details of the function **c_quorum_rpc** are elided; it issues requests to the preferred quorum of servers and, if necessary, probes additional servers until a quorum of responses is received. Because of the crash-recovery failure model and unreliable channels, **c_quorum_rpc** repeatedly sends requests until a quorum of responses is received. Server responses include a status (success or failure), an answer, and a replica history (see lines 1008, 1021, and 1030 of **s_request**). From the quorum of responses, those that indicate success are counted to determine the order of the candidate. The answer corresponding to the candidate is also identified, and the object history set is updated with the returned replica histories.

If an operation does not successfully complete, then repair is performed. Repair involves performing barrier and copy operations until classification identifies a complete object version that can be conditioned on. The function **c_repair** performs repair. To promote progress, **backoff** is called (line 502, pseudo-code not shown). Note that an operation that does not complete because its conditioned-on OHS is not current may not need to perform repair: after a client updates its OHS with server responses, classification may indicate that the client perform a method. This happens if a client's cached OHS is out of date and the object is accessed contention-free. Classification of the OHS dictates whether a barrier or a copy is performed. In both cases, the client issues a **c_quorum_rpc** with the OHS. Once a copy successfully completes, a method is allowed to be performed and repair returns.

```
structures, types, & enumerations:
100: Class ∈ {QUERY, UPDATE}                          /* Enumeration. */
101: Type ∈ {METHOD, COPY, BARRIER}                   /* Enumeration. */
102: Operation ≡ ⟨Method, Class,Argument⟩
103: LT ≡ ⟨Time, BarrierFlag, ClientID, Operation,OHS⟩
104: Candidate ≡ ⟨LT, $LT_{CO}$⟩                      /* Pair of timestamps. */
105: RH ≡ {Candidate}                         /* Ordered set of candidates. */
106: α ≡ HMAC[U]                               /* Array indexed by server. */
107: OHS ≡ ⟨RH, α⟩ [U]                         /* Array indexed by server. */

c_initialize() : /* Client initialization. */
200: ∀s ∈ U,OHS[s].RH := ⟨0, 0⟩,OHS[s].α := ⊥

c_ increment(Argument): /* Example update operation. */
300: Operation := ⟨increment, UPDATE,Argument⟩
301: ⟨Answer, Order,OHS⟩ := c_quorum_ rpc(Operation,OHS)
302: while (Order < q) do
303:             /* Repair and retry update until candidate established. */
304:     c_repair(OHS)
305:     ⟨Answer, Order,OHS⟩ := c_quorum rpc(Operation,OHS)
306: end while
307: return (⟨Answer⟩)

c_fetch():                                    /* Example query operation. */
400: Operation := ⟨fetch, QUERY, ⊥⟩
401: ⟨Answer, Order,OHS⟩ := c_quorum_ rpc(Operation,OHS)
402: while (Order < q) do
403:     c_repair(OHS)
404:     ⟨Answer, Order,OHS⟩ := c_quorum rpc(Operation,OHS)
405: end while
406: return (⟨Answer⟩)

c_repair(OHS):                       /* Deal with failures and contention. */
500: ⟨Type, ⊥, ⊥⟩ := classify(OHS)
501: while (Type ≠ METHOD) do
502:     backoff()                          /* Backoff to avoid livelock. */
503:     /* Perform a barrier or copy (depends on OHS). */
504:     ⟨⊥,⊥,OHS⟩ := c_quorum_rpc(⊥,OHS)
505:     ⟨Type, ⊥, ⊥⟩ := classify(OHS)
506: end while
507: return

c_quorum_rpc(Operation,OHS) :                             /* Quorum RPC. */
600: /* Eliding details of sending to/probing for a quorum. */
601: /* Get quorum of s_request(Operation,OHS) responses. */
602: /* For each response, update OHS[s] based on s.RH. */
603: /* Answer and Order come from successful responses. */
604: return (⟨Answer, Order,OHS⟩)

classify(OHS):                /* Classify candidate in object history set. */
700: /* Determine latest object version and barrier version. */
701: ObjCand := latest_candidate(OHS, FALSE)
702: BarCand := latest_candidate(OHS, TRUE)
703: $LT_{lat}$ := latest_time(OHS)
704: /* Determine which type of operation to perform. */
705: Type := BARRIER                                 /* Default operation. */
706: /* If an established barrier is latest, perform COPY. */
707: if ($LT_{lat}$ = BarCand.LT) ^ (order(BarCand,OHS) ≥ q)
708:     then Type := COPY
709: /* If an established object is latest, perform METHOD. */
710: if ($LT_{lat}$ = ObjCand.LT) ^ (order(ObjCand,OHS) ≥ q)
711:     then Type := METHOD
712: return (⟨Type, ObjCand, BarCand⟩)

order(Candidate,OHS) :                  /* Determine order of candidate. */
800: return (|{s ∈ U : Candidate ∈ OHS[s].RH}|)

s_initialize() :  /* Initialize server s. */
900: s.RH := {⟨0, 0⟩}
901: ∀s?∈ U, s.α[s?] := hmac(s, s?, s.RH)

s_request(Operation,OHS) :                /* Handle request at server s. */
1000: Answer := ⊥                         /* Initialize answer to return. */
1001: ∀s?∈ U, if (hmac(s, s?,OHS[s?].RH) ≠ OHS[s?].α[s])
1002:     then OHS[s?].RH := {⟨0, 0⟩}             /* Cull invalid RHs. */
1003: /* Setup candidate based on Operation and OHS. */
1004: ⟨Type, ⟨LT, $LT_{CO}$⟩, $LT_{current}$⟩ := s_setup(Operation,OHS)
1005: /* Eliding details of receiving same request multiple times. */
1006: /* Determine if OHS is current (return if not). */
1007: if (latest_time(s.RH) > $LT_{current}$)
1008:     then reply(s, FAIL, ⊥, s. ⟨RH, α⟩)
1009: /* Retrieve conditioned-on object version. */
1010: if (Type ∈ {METHOD, COPY}) then
1011:     Object := retrieve($LT_{CO}$)
1012:     /* Object sync if object version not stored locally. */
1013:     if ((Object = ⊥) ^ ($LT_{CO}$ > 0))
1014:         then Object := object_sync($LT_{CO}$)
1015: end if
1016: /* Perform operation on conditioned-on object version. */
1017: if (Type = METHOD) then
1018:     ⟨Object,Answer⟩ :=                            /* Invoke method. */
1019:         Operation.Method(Object, Operation.Argument)
1020:     if (Operation.Class = QUERY)
1021:         then reply(s, SUCCESS, Answer, s.⟨RH, α⟩)
1022: end if
1023: /* Update server state to include new object version. */
1024: atomic
1025:     s.RH := s.RH ∪ {⟨LT, $LT_{CO}$⟩}
1026:     ∀s?∈ U, s.α[s?] := hmac(s, s?, s.RH)
1027:     store(LT, Object)
1028:     /* Could prune replica history. */
1029: end atomic
1030: reply(s, SUCCESS, Answer, s. ⟨RH, α⟩)

s_setup(Operation,OHS) :                             /* Setup candidate. */
1100: ⟨Type, ObjCand, BarCand⟩ := classify(OHS)
1101: $LT_{CO}$ = ObjCand.LT
1102: LT.Time := latest_time(OHS).Time + 1
1103: LT.ClientID := ClientID
1104: LT.OHS := OHS
1105: if (Type = METHOD) then
1106:     LT.BarrierFlag := FALSE
1107:     LT.Operation := Operation
1108:     $LT_{current}$ := ObjCand.LT         /* (= $LT_{CO}$) cf. line 1101 */
1109: else if (Type = BARRIER) then
1110:     LT.BarrierFlag := TRUE
1111:     LT.Operation := ⊥
1112:     $LT_{current}$ := LT
1113: else
1114:     LT.BarrierFlag := FALSE                          /* Type = COPY */
1115:     LT.Operation := $LT_{CO}$.Operation
1116:     $LT_{current}$ := BarCand.LT
1117: end if
1118: return (Type, ⟨LT, $LT_{CO}$⟩, $LT_{current}$⟩)

latest_candidate(OHS, BarrierFlag)
1200: /* Set of all repairable/complete objects (barriers). */
1201: CandidateSet := {Candidate : (order(Candidate,OHS) ≥ r)
1202:     ^(Candidate.LT.BarrierFlag = BarrierFlag)}
1203: /* Repairable/complete Candidate with max timestamp. */
1204: Candidate := (Candidate : (Candidate ∈ CandidateSet)
1205:     ^(Candidate.LT = max(CandidateSet.LT)))
1206: return (Candidate)
```

Figure 4.3: Query/Update pseudo-code. Client functions are prefixed with c_ and server functions with s_. The following symbols are used in the pseudo-code: s (server), U (the universe of servers), OHS (object history set), RH (replica history), α (authenticator), LT (timestamp), LT_{CO} (conditioned-on timestamp), 0 (initial logical time), ⊥ (null value), q (the threshold for classifying a candidate complete), and r (the threshold for classifying a candidate repairable).

The function **classify** classifies an object history set: it identifies what type of operation (method, barrier, or copy) must be performed and the timestamps of the latest object and barrier versions. Clients and servers both call **classify**: a client calls it to determine the action dictated by its cached OHS and a server calls it to determine the action dictated by the conditioned-on OHS sent from a client. The function **latest_candidate** determines the latest candidate that is classified as either repairable or complete in the conditioned-on OHS is latest. It is called to identify the latest object version (line 701) and latest barrier version (line 702). The function **latest_time** (pseudo-code not shown) is called to determine the latest timestamp in the conditioned-on OHS. If the latest timestamp in the conditioned-on OHS is greater than the timestamps of the latest object and barrier versions, then the latest candidate is incomplete and a barrier must be performed (line 705). If the latest timestamp matches either the latest object or barrier version, but it is not classified as complete, then a barrier is performed. If the latest candidate is a complete barrier version, then a copy is performed (line 708). If the latest candidate is a complete object version, then a query or update is performed, conditioned on it (line 711).

Server initialization illustrates how authenticators are constructed via the **hmac** function (line 901). Each entry in an authenticator is an HMAC (keyed hash) over the server's replica history [Bellare96]. The two servers passed into **hmac** identify the shared key for taking the HMAC.

The function **s_request** processes requests at a server. First, the server validates the integrity of the conditioned-on OHS. Any replica histories with invalid authenticators are set to null. Next, the server calls s setup to setup the candidate. It calls **classify** (line 1100) and sets various parts of the logical timestamp accordingly. In performing classification on the OHS, the server determines whether to perform a barrier, a copy, or a method operation. Given the setup candidate, the server determines if the object history set is current. If it is, the server attempts to fetch (via **retrieve** on line 1011) the conditioned-on object version from its local store. If it is not stored locally, the conditioned-on object version is retrieved from other servers (via **object_sync**, pseudo-code not shown, on line 1014). For queries and updates, the server invokes a method on the conditioned-on object version. If the method invoked is a query, then the server returns immediately, because the server state is not modified. If the method invoked is an update, then a new object version results. For update methods and for copy methods, the server adds the new candidate to its replica history, updates its authenticator, and locally stores (via **store**) the new object version indexed by timestamp. These three actions (lines 1025-1027) are performed atomically to be correct in the crash-recovery failure model. Finally, the server returns the answer and its updated replica history and authenticator.

4.3.5 Implementation Details

This section discusses interesting aspects of system design, as well as implementation details and optimizations omitted from the pseudo-code. Cached object history set.

Clients cache object history sets of objects they access. After a client's first access, the client performs operations without first requesting replica histories. If the cached object history set is current, then the operation completes in a single phase. If not, then a server rejects the operation and returns its current replica history so that the client can make its object history set more current.

Optimistic query execution. If a client has not accessed an object recently (or ever), its cached OHS may not be current. It is still possible for a query to complete in a single phase. Servers, noting that the conditioned-on OHS is not current, invoke the query method on the latest object version they store. The server, in its response, indicates that the OHS is not current, but also includes the answer for the query invoked on the latest object version and its replica history. After the client has received a quorum of server responses, it performs classification on its object history set. Classification allows the client to determine if the query was invoked on the latest complete object version; if so the client returns the answer.

This optimization requires additional code in the function **s_request**, if the condition on line 1007 is true. To optimistically execute the query, the server retrieves the latest object version in its replica history and invokes the query on that object version (as is done on lines 1018 and 1019).

Quorum access strategy and probing. In traditional quorum-based protocols, clients access quorums randomly to minimize per-server load. In contrast, in the Q/U protocol, clients access an object's preferred quorum. Clients initially send requests to the preferred quorum. If responses from the preferred quorum are not received in a timely fashion, the client sends requests to additional servers. Accessing a non-preferred quorum requires that the client probe to collect a quorum of server responses. Servers contacted that are not in the preferred quorum will likely need to object sync.

In our system, all objects have an ID associated with them. In the current implementation, a deterministic function is used to map an object's ID to its preferred quorum. A simple policy of assigning preferred quorums based on the object ID modulo *n* is employed. Assuming objects are uniformly loaded (and object IDs are uniformly distributed), this approach uniformly loads servers.

The probing strategy implemented in the prototype is parameterized by the object ID and the set of servers which have not yet responded. Such a probing strategy maintains locality of quorum access if servers crash and disperses load among non-crashed servers.

Quorum-based techniques are often advocated for their ability to disperse load among servers and their throughputs-scalability (e.g., [Naor98, Wool98, Malkhi00b]). Preferred quorums, because of the locality of access they induce, do not share in these properties if all operations are performed on a set of objects with a common preferred quorum. To address such a concern, the mapping between objects and preferred quorums could be made dynamic. This would allow traditional load balancing techniques to be employed that "migrate" loaded objects to distinct preferred quorums over time.

Inline repair. Repair requires that a barrier and copy operation be performed. *Inline* repair does not require a barrier or a copy; it repairs a candidate "in place" at its timestamp by completing the operation that yielded the candidate. Inline repair operations can thus complete in a single round trip. Inline repair is only possible if there is no *contention* for the object. Any timestamp in the object history set greater than that of the repairable candidate indicates contention. A server processes an inline repair request in the same manner as it would have processed the original update for the candidate (from the original client performing the update). Inline repair can also be performed on barrier candidates. Inline repair is useful in the face of server failures that lead to non-preferred quorum accesses. The first time an object is accessed after a server in its preferred quorum has failed, the client can likely perform inline repair at a non-preferred quorum.

Handling repeated requests at the server. A server may receive the same request multiple times. Two different situations lead to repeated requests. The crash-recovery failure model requires clients to repeatedly send requests to ensure a response is received. As such, servers may receive the same request from the same client multiple times. Additionally, inline repair may lead to different clients (each performing inline repair of the same candidate) sending the same request to the server. Regardless of how many times a server receives a requests, it must only invoke the method on the object once. Moreover, the server should respond to a repeated request in a similar manner each time. As such, it is necessary for the server to store the answer to update operations with the corresponding object version. This allows a server to reply with the same answer each time it receives a repeated request. Before checking if the conditioned-on OHS is current, the server checks to determine if a candidate is already in its replica history with the timestamp it setup (cf. line 1005). If so, the server retrieves the corresponding answer and returns immediately.

Retry and backoff policies. Update-update concurrency among clients leads to contention for an object. Clients must backoff to avoid livelock. A random exponential backoff policy is implemented (cf. [Chockler01]). Update-query concurrency does not necessarily lead to contention. A query concurrent to a single update may observe a repairable or incomplete candidate. In the case of the former, the client performs an inline repair operation; such an operation does not contend with the update it is repairing (since servers can accept the same request multiple times). In the case of the latter, additional client logic is required, but it is possible to invoke a query on the latest established candidate, in spite of there being later incomplete candidates. On the client-side, after a query operation is deemed to have failed, additional classification is performed. If classification identifies that all of the candidates with timestamps later then the latest established candidate are incomplete, then the answer from invoking the query on the established candidate is returned. This situation can arise if an update and query are performed concurrently. Because of inline repair and this optimization, an object that is updated by a single client and queried by many other clients does not ever require barriers, copies, or backoff.

Object syncing. To perform an object sync, a server uses the conditioned-on object history set to identify b+1 servers from which to solicit the object state. Like accessing a preferred quorum, if b+1 responses are not forthcoming (or do not all match) additional servers are probed.

It is possible to trade-off network bandwidth for server computation: only a single correct server need send the entire object version state and other servers can send a collision-resistant hash of the object version state. A different approach is to include the hash of the object state in the timestamp. This requires servers to take the hash of each object version they create. However, it allows servers to contact a single other server to complete object syncing or for the object state to be communicated via the client.

Authenticators. Authenticators consist of n HMACs, one for each server in the system. Authenticators are taken over the hash of a replica history, rather than over the entire replica history. The use of n HMACs in the authenticator, rather than a digital signature, is based on the scale of systems currently being evaluated. The size and computation cost of digital signatures do not grow with n. As such, if n were large enough, the computation and network costs of employing digital signatures would be less than that of HMACs.

Compact timestamps. The timestamps in the pseudocode are quite large, since they include the operation and the object history set. In the implementation, a single hash over the operation and the object history set replaces these elements of logical timestamps. The operation and object history set are necessary to uniquely place a specific operation at a single unique point in logical time. Replacing these elements with a collision resistant hash serves this purpose and improves space-efficiency.

Compact replica histories. Servers need only return the portion of their replica histories with timestamps greater than the conditioned-on timestamp of the most recent update accepted by the server. As such, a server *prunes* its replica history based on the conditioned-on timestamp after it accepts an update request. (Although pruning pseudocode is not shown, pruning would occur on line 1028.) In the common case, the replica history contains two candidates: one for the latest object version and one for the conditioned-on object version. Failures and concurrency may lead to additional barrier and incomplete candidates in the replica history.

Object versions corresponding to candidates pruned from a server's replica history could be deleted. Doing so would reclaim storage space on servers. However, deleting past object versions could make it impossible for a server to respond to slow repeated requests from clients and slow object sync requests from other servers. As such, there is a practical trade-off between reclaiming storage space on servers and ensuring that clients do not need to abort operations.

Malevolent components. As presented, the pseudo-code for the Q/U protocol ensures safety, but not progress, in the face of malevolent components. Malevolent components can affect the ability of correct clients to make progress. However, the Q/U protocol can be modified to ensure that isolated correct clients can make progress.

Malevolent clients may not follow the specified backoff policy. For contended objects, this is a form of denial-of-service attack. However, additional server-side code could rate limit clients.

Malevolent clients may issue updates only to subsets of a quorum. This results in latent work in the system, requiring correct clients to perform repair. Techniques such as *lazy verification* [Abd-El-Malek05a], in which the correctness of client operations is verified in the background and in which limits are placed on the amount of unverified work a client may inject in the system, can bound the impact that such malevolent clients have on performance.

A malevolent server can return an invalid authenticator to a client. Note that a client cannot validate any HMACs in an authenticator and each server can only validate a single HMAC in an authenticator. On line 1002, servers cull replica histories with invalid authenticators from the conditioned-on OHS. Servers return the list of replica histories (i.e., the list of invalid authenticators) culled from the conditioned-on OHS to the client. A server cannot tell whether a malevolent client or server corrupted the authenticator. A malevolent server can also return lists of valid authenticators to the client, indicating they are invalid. If a malevolent server corrupts more than *b* HMACs in its authenticator, then a client can conclude, from the responses of other servers, that the server is malevolent and exclude it from the set of servers it contacts. If a malevolent server corrupts fewer than *b* HMACs in its authenticator, then a client can conclude only that some servers in the quorum contacted are malevolent. This is sufficient for the client to solicit more responses from additional servers and make progress.

A malevolent server can reject a client update for not being current by forging a candidate with a higher timestamp in its replica history. Due to the constraints on the quorum system and the classification rules, malevolent servers cannot forge a potential or established candidate: any forged candidate is classifiable as incomplete. As such, the client must perform repair (i.e., a barrier then a copy). Consider a client acting in isolation. (Recall that contending clients can lead to justifiably rejected updates.) If an isolated client has its update accepted at all benign servers in a quorum, it by definition establishes a candidate (even though the client may not be able to classify the candidate as complete). Given that a server rejected its update, the client initiates repair with a barrier operation. Malevolent servers can only reject the isolated client's barrier operation by forging a barrier candidate with a higher timestamp. Any other action by the malevolent server is detectable as incorrect by the client. The Q/U protocol, as described in the pseudocode, allows a malevolent server to keep generating barriers with higher timestamps. To prevent malevolent servers from undetectably repeatedly forging barrier candidates, classification is changed so that any barrier candidate not classified as complete is inline repaired. This bounds the number of barriers that can be accepted before a copy must be accepted to be the number of possible distinct incomplete candidates in the system (i.e., *n*). With such a modification, to remain undetected, a malevolent server must either accept the copied candidate or not respond to the client; either action allows the client to make progress. Since only potential candidates can be copied forward and potential candidates must intersect established candidates, malevolent servers cannot undetectably forge a copied candidate (given an established candidate prior to the barriers).

In summary, with the modifications outlined here, the Q/U protocol can ensure that isolated clients make progress in spite of malevolent components.

Pseudo-code and quorums. Extended pseudo-code is available in a companion technical report [Abd-El-Malek05b]. The extended pseudo-code includes optimistic query execution, additional client-side classification for queries to reduce query-update contention, inline repair of value candidates and barrier candidates, handling repeated requests at the server, pruning replica histories, deleting obsolete object versions, and object syncing.

The pseudo-code is tailored for threshold quorums to simplify its presentation. For general quorums that meet the constraints given in 4.3.3, changes to the pseudo-code are localized. The functions **c_quorum_rpc** and **order** must change to handle general quorums rather than threshold quorums. Any tests of whether a candidate is repairable or complete must also change to handle general quorums (e.g., lines 302, 402, 707, 710, and 1201).

4.3.6 Example Q/U Protocol Execution

An example execution of the Q/U protocol is given in Table 4.1. The caption explains the structure of, and notation used in, the table. The example is for an object that exports two methods: **get** (a query) and **set** (an update). The object can take on four distinct values (♣,♦,♥,♠) and is initialized to ♥. The server configuration is based on the smallest quorum system for $b = 1$. Clients perform optimistic queries for the **get** method; the conditioned-on OHS sent with the **set** method is not shown in the table. The sequence of client operations is divided into four sequences of interest by horizontal double lines. For illustrative purposes, clients *X* and *Z* interact with the object's preferred quorum (the first five servers) and *Y* with a non-preferred quorum (the last five servers).

The first sequence demonstrates failure- and concurrency-free execution: client *X* performs a **get** and **set** that each complete in a single phase. Client *Y* performs a get in the second sequence that requires repair. Since there is no contention, *Y* performs inline repair. Server s_5 performs object syncing to process the inline repair request.

In the third sequence, concurrent updates by *X* and *Y* are attempted. Contention prevents either update from completing successfully. At server s_4, the **set** from *Y* arrives before the **set** from *X*. As such, it returns its replica history $\{\langle 3,1\rangle,\langle 1,\mathbf{0}\rangle\}$ with FAIL. The candidate $\langle 3,1\rangle$ in this replica history dictates that the timestamp of the barrier be 4**b**. For illustrative purposes, *Y* backs off. Client *X* subsequently completes barrier and copy operations.

In the fourth sequence, *X* crashes during a **set** operation and yields a potential candidate. The remaining clients *Y* and *Z* perform concurrent **get** operations at different quorums; this illustrates how a potential candidate can be classified as either repairable or incomplete. *Y* and *Z* concurrently attempt a barrier and inline repair respectively. In this example, *Y* establishes a barrier before *Z*'s inline repair requests arrive at servers s_3 and s_4. Client *Z* aborts its inline repair operation. Subsequently, *Y* completes a copy operation and establishes a candidate $\langle 8, 5\rangle$ that copies forward the established candidate $\langle 5; 2\rangle$ Notice that the replica histories returned by the servers in response to the **get** requests are pruned. For example, server s0 returns $\{\langle 6; 5\rangle; \langle 5; 2\rangle\}$, rather than $\{\langle 6, 5\rangle, \langle 5, 2\rangle, \langle 4\mathbf{b}, 2\rangle, \langle 2, 1\rangle, \langle 1, \mathbf{0}\rangle, \langle \mathbf{0}, \mathbf{0}\rangle\}$, because the object history set sent by *X* with **set** operation $\langle 6, 5\rangle$ proved that $\langle 5, 2\rangle$ was established.

4.3.7 Multi-object Updates

We promote the decomposition of services into objects to promote concurrent non-contending accesses. The Q/U protocol allows some updates to span multiple objects. A multi-object update atomically updates a set of objects. To perform such an update, a client includes a conditioned-on OHS for each object being updated. The set of objects and each object's corresponding conditioned-on OHS, are referred to as the multi-object history set (*multi-OHS*). Each server locks its local version of each object in the multi- OHS and validates that each conditioned-on OHS is current. The local locks are acquired in object ID order to avoid the possibility of local deadlocks. Assuming validation passes for all of the objects in the multi-OHS, the server accepts the update for all the objects in the multi-OHS simultaneously.

So long as a candidate is classified as complete or incomplete, no additional logic is required. However, to repair multi-object updates, clients must determine which objects are in the multi-OHS. As such, the multi-OHS is included in the timestamp. Note that all objects updated by a multi-object update have the same multi-OHS in their timestamp.

To illustrate multi-object repair, consider a multi-object update that updates two objects, o_a and o_b. The multi-object update results in a candidate for each object, c_a and c_b respectively. Now, consider a client that queries o_a and classifies c_a as repairable. To repair c_a, the client must fetch a current object history for o_b because o_b is in the multi-OHS of c_a. If c_a is in fact established, then c_b is also established and there could exist a subsequent established candidate at o_b that conditions on c_b. If c_a is not established, then c_b also is not established and subsequent operations at o_b may preclude c_b from ever being established (e.g., a barrier and a copy that establishes another candidate at o_b with a higher timestamp than c_b). The former requires that c_a be reclassified as complete. The latter requires that c_a be reclassified as incomplete.

Operation	s_0	s_1	s_2	s_3	s_4	s_5	**Result**
Initial system	⟨**0**,**0**⟩,♥	⟨**0**,**0**⟩,♥	⟨**0**,**0**⟩,♥	⟨**0**,**0**⟩,♥	⟨**0**,**0**⟩,♥	⟨**0**,**0**⟩,♥	
X completes **get**()	{**0**},♥	{**0**},♥	{**0**},♥	{**0**},♥	{**0**},♥		⟨**0**,**0**⟩ complete, return ♥
X completes **set**(♣)	⟨1,**0**⟩,♣	⟨1,**0**⟩,♣	⟨1,**0**⟩,♣	⟨1,**0**⟩,♣	⟨1,**0**⟩,♣		⟨1,**0**⟩ established
Y begins **get**()…		{1,**0**},♣	{1,**0**},♣	{1,**0**},♣	{1,**0**},♣	{**0**},♥	⟨1,**0**⟩ repairable
…*Y* performs **inline**						{1,**0**},♣	⟨1,**0**⟩ complete, return ♣
X attempts set(♦)… *Y* attempts set(♥)	⟨2,1⟩,♦	⟨2,1⟩,♦ FAIL	⟨2,1⟩,♦ FAIL	⟨2,1⟩,♦ FAIL	FAIL ⟨3,1⟩,♥	⟨3,1⟩,♥	⟨2,1⟩ potential *Y* backs off
…*X* completes **barrier**	⟨4**b**,2⟩,⊥	⟨4**b**,2⟩,⊥	⟨4**b**,2⟩,⊥	⟨4**b**,2⟩,⊥	⟨4**b**,2⟩,⊥		⟨4**b**,2⟩ established
…*X* completes **copy**	⟨5,2⟩,♦	⟨5,2⟩,♦	⟨5,2⟩,♦	⟨5,2⟩,♦	⟨5,2⟩,♦		⟨5,2⟩ established
X crashes in **set**(♠)	⟨6,5⟩,♠	⟨6,5⟩,♠	⟨6,5⟩,♠				⟨6,5⟩ potential
Y begins **get**()… *Z* begins **get**()…	 {6,5},♠	{6,5},♠ {6,5},♠	{6,5},♠ {6,5},♠	{5,4**b**,2,1},♦ {5,4**b**,2,1},♦	{5,4**b**,2,1},♦ {5,4**b**,2,1},♦	{3,1},♥	⟨6,5⟩ inc., ⟨5,2⟩ rep. ⟨6,5⟩ repairable
…*Y* completes **barrier** …*Z* attempts **inline**		⟨7**b**,5⟩,⊥	⟨7**b**,5⟩,⊥	⟨7**b**,5⟩,⊥ FAIL	⟨7**b**,5⟩,⊥ FAIL	⟨7**b**,5⟩,⊥	⟨7**b**,5⟩ established *Z* backs off
…*Y* completes **copy**		⟨8,5⟩,♦	⟨8,5⟩,♦	⟨8,5⟩,♦	⟨8,5⟩,♦	⟨8,5⟩,♦	⟨8,5⟩ est., return ♦

Table 4.1: Example Q/U protocol execution. Operations performed by three clients (*X, Y*, and *Z*) are listed in the left column. The middle columns list candidates stored by and replies (replica histories or status codes) returned by six benign servers (s_0,…,s_5). The right column lists if updates yield an established or potential candidate (assuming all servers are benign) and the results of classification for queries. Time "flows" from the top row to the bottom row. Candidates are denoted ⟨*LT*, LT_{CO}⟩. Only *LT.Time* with "b" appended for barriers is shown for timestamps (i.e., client ID, operation, and object history set are not shown). Replica histories are denoted {LT_3, LT_2, …}. Only the candidate's *LT*, not the LT_{CO}, is listed in the replica history.

Such reclassification of a repairable candidate, based on the objects in its multi-OHS, is called *classification by deduction*. If the repairable candidate lists other objects in its multi-OHS, then classification by deduction must be performed. If classification by deduction does not result in reclassification, then repair is performed. Repair, like in the case of individual objects, consists of a barrier and copy operation. The multi-OHS for multi-object barriers and multi-object copies have the same set of objects in them as the multi-OHS of the repairable candidate. Because multi-object operations are atomic, classification by deduction cannot yield conflicting reclassifications: either all of the objects in the multi-OHS are classified as repairable, some are classified complete (implying that all are complete), or some are classified incomplete (implying that all are incomplete).

Multi-object updates may span objects with different preferred quorums. The preferred quorum for one of the objects involved is selected for the multi-object update. In our implementation, we use the preferred quorum of the highest object ID in the multi-OHS. As such, some servers will likely have to object sync for some of the objects. Moreover, some of the candidates a multi-object write establishes may be outside of their preferred quorums. If there is no contention, such candidates are repairable at their preferred quorum via inline repair.

4.3.8 Correctness

This section discusses, at a high level, the safety and liveness properties of the Q/U protocol. In the companion technical report [Abd-El-Malek05b], the safety and liveness of the Q/U protocol is discussed in more detail, extended pseudo-code is provided, and a proof sketch of safety is given for a variant of the Q/U protocol, the Read/Conditional-Write protocol.

In the Q/U protocol, operations completed by correct clients are *strictly serializable* [Bernstein87]. Operations occur atomically, including those that span multiple objects, and appear to occur at some point in time between when the operation begins and some client observes its effect. If the Q/U protocol is used only to implement individual objects, then correct clients' operations that complete are linearizable [Herlihy90].

To understand how the Q/U protocol ensures strict serializability, consider the *conditioned-on chain*: the set of object versions that are found by traversing back in logical time from the latest established candidate via the conditioned-on timestamp. Every established object version is in the conditioned-on chain and the chain goes back to the initial candidate $\langle \mathbf{0}, \mathbf{0} \rangle$. The conditioned-on chain induces a total order on update operations that establish candidates, which ensures that all operations are strictly serializable. The conditioned-on chain for the example system execution in Table 4.1 is $\langle 8, 5 \rangle \rightarrow \langle 5, 2 \rangle \rightarrow \langle 2, 1 \rangle \rightarrow \langle 1, \mathbf{0} \rangle \rightarrow \langle \mathbf{0}, \mathbf{0} \rangle$. Given the crash-recovery server fault model for benign servers, progress cannot be made unless sufficient servers are up. This means that services implemented with the Q/U protocol are not available during network partitions, but become available and are correct once the network merges. The liveness property of the Q/U protocol is fairly weak: it is possible for clients to make progress (complete operations).

Under contention, operations (queries or updates) may abort so it is possible for clients to experience livelock. In a benign execution—an execution without malevolent clients or servers—the Q/U protocol is *obstruction-free* [Herlihy03]: an isolated client can complete queries and updates in a finite number of steps. The extensions sketched in Section 4.3.5 also allow an isolated client to complete queries and updates in a finite number of steps in an execution with malevolent servers.

4.4 Evaluation

This section evaluates the Q/U protocol as implemented in our prototype library. First, it compares the fault-scalability of a counter implemented with the Q/U protocol and one implemented with the publicly available[2] Byzantine fault-tolerant, agreement-based BFT library [Castro02]. Second, it quantifies the costs associated with the authenticator mechanism and with non-optimal object accesses in the Q/U protocol. Third, it discusses the design and performance of an NFSv3 metadata service built with the Q/U protocol.

4.4.1 Experimental Setup

The Q/U protocol prototype is implemented in C/C++ and runs on both Linux and Mac OS X. Client-server communication is implemented via RPCs over TCP/IP sockets. MD5 cryptographic hashes are employed [Rivest92]; the authors recognize that the MD5 hash is showing its age [Wang04], however its use facilitates comparisons with other published results for BFT. All experiments are performed on a rack of 76 Intel Pentium 4 2.80 GHz computers, each with 1GB of memory, and an Intel PRO/1000 NIC. The computers are connected via an HP ProCurve Switch 4140gl with a specified internal maximum bandwidth of 18.3 Gbps (or 35.7 mpps). The computers run Linux kernel 2.6.11.5 (Debian 1:3.3.4-3). Experiments are run for 30 seconds and measurements are taken during the middle 10 seconds; experiments are run 5 times and the mean is reported. The standard deviation for all results reported for the Q/U prototype is less than 3% of the mean (except, as noted, for the contention experiment). The working sets of objects for all experiments fit in memory, and no experiments incur any disk accesses.

4.4.2 Fault-scalability

We built a prototype counter object with the Q/U protocol and with the BFT library. The **increment** method, an update (read-write operation in BFT terminology), increments the counter and returns its new value. The **fetch** method, a query, simply returns the current value of the counter. A counter object, although it seems simple, requires the semantics these two protocols provide to correctly increment the

[2] http://www.pmg.lcs.mit.edu/bft/#sw

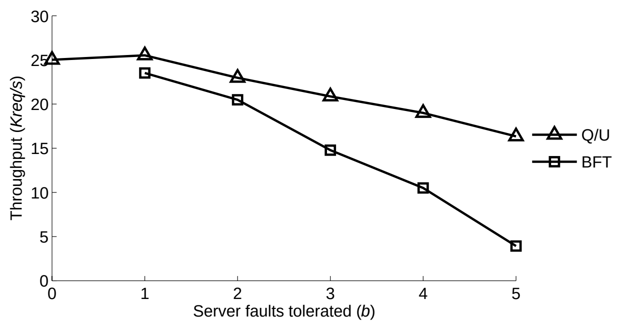


Figure 4.4: Fault-scalability.

current value. Moreover, the simplicity of the object allows us to focus on measuring the inherent network and computation costs of these Byzantine fault-tolerant protocols.

We measure the throughput (in requests per second) of counter objects as the number of malevolent servers tolerated increases. Since both the Q/U protocol and BFT implement efficient, optimistic queries (read-only in BFT terminology), we focus on the cost of updates. These experiments are failure-free. For the Q/U-based counter, clients access counter objects via their preferred quorum and in isolation (each client accesses a different set of counter objects). We ran a single instance of the BFT-based counter so that there is a single primary replica that can make effective use of batching (an optimization discussed below). As such, this experiment compares best case performance for both protocols.

Figure 4.4 shows the fault-scalability of the Q/U-based and BFT-based counters. The throughput of **increment** operations per second, as the number of server faults tolerated increases from $b = 0$ to $b = 5$, is plotted. No data point is shown for the BFT-based counter at $b = 0$, a single server that tolerates no failures, because we had difficulties initializing the BFT library in this configuration. The increase in throughput from $b = 0$ to $b = 1$ for the Q/U-based counter is due to quorum throughput-scalability: at $b = 1$, each server processes only five out of every six update requests.

The data points shown in Figure 4.4 correspond to the peak throughput observed. To determine the peak throughput, we ran experiments with one physical client, then three physical clients, and so on, until we ran with thirty-three physical clients. Two client processes ran on each physical client; in all cases, this was sufficient to load both the Q/U-based and BFT-based counters. The throughput plotted for each value of b corresponds to that for the number of clients that exhibited the best throughput averaged over five runs. The BFT-based counter is less well-behaved under load than the Q/U-based counter: for each value of b, the performance of the BFT-based counter depends significantly on load. Throughput increases as load is added until peak throughput is achieved, then additional load reduces the observed throughput dramatically. For the Q/U-based counter, in contrast, peak throughput is maintained as additional load is applied. Due to the behavior of BFT under load, we report the peak throughput rather than the throughput observed for some static number of clients. The Q/U-based counter provides higher throughput than the BFT-based counter in all cases. Both counters provide similar throughput when tolerating one or two faults. However, because of its fault-scalability, the Q/U-based counter, provides significantly better throughput than the BFT-based counter as the number of faults tolerated increases. When compared to the BFT-based counter, the Q/U-based counter provides 1990 more requests per second at $b = 1$ and 12400 more requests per second at $b = 5$. As b is increased from 1 to 5, the performance of the Q/U counter object degrades by 36%, whereas the performance of the BFT-based counter object degrades by 83%.

In BFT, server-to-server broadcast communication is required to reach agreement on each batch of requests. In batching requests, BFT amortizes the cost of generating and validating authenticators over several requests. For low values of b, batching is quite effective (fifteen or more requests per batch). As the number of faults tolerated increases, batching becomes less effective (just a few requests per batch at $b = 5$). Moreover, the cost of constructing and validating authenticators grows as the number of faults tolerated increases. The cost of authenticators grows slower for BFT than for the Q/U protocol, since BFT requires fewer servers to tolerate a given number of faults (as is illustrated in Figure 4.1, $3b + 1$ rather than $5b + 1$ servers).

BFT uses multicast to reduce the cost of server-to-server broadcast communication. However, multicast only reduces the number of messages servers must send, not how many they must receive (and authenticate). Since BFT employs multicast, all communication is based on UDP. We believe that the backoff and retry policies BFT uses with UDP multicast are not well-suited to high-throughput services.

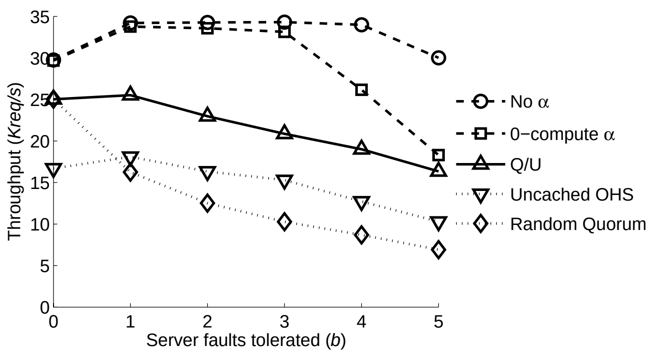


Figure 4.5: Details of Q/U protocol fault-scalability.

$\frac{5b+1}{4b+1}\times$

4.4.3 Fault-scalability Details

Authenticators and fault-scalability. Figure 4.1 in Section 4.2 illustrates the analytic expected throughput for the threshold quorums employed in the Q/U protocol prototype: the throughput provided by a single server. Our measurements for the Q/U-counter object in Figure 4.4 do not match that expectation. Throughput of the Q/U-counter declines (slowly) as the number of faults tolerated increases beyond one. The decline is due to the cost of authenticators.

As the number of faults tolerated increases, authenticators require more server computation to construct and validate, as well as more server bandwidth to send and receive. At $b = 1$, it takes 5.6 µs to construct or validate an authenticator, whereas at b = 5, it takes 17 µs. In Figure 4.5, we show the fault-scalability of the Q/U protocol for two hypothetical settings: one in which authenticators are not used at all (the "No α" line) and one in which HMACs require no computation to construct or validate (the "0-compute α" line). Compare these results with the "Q/U" line (the line also shown in Figure 4.4). The results for No α demonstrate the best fault-scalability that the Q/U protocol can achieve on our experimental infrastructure. Dedicated cryptographic processors or additional processing cores could someday realize the 0-compute α results. If the server computation required to construct and validate authenticators could be offloaded, the Q/U protocol would exhibit near ideal faults-scalability.

Cached object history sets. Figure 4.5 also shows the cost of accessing objects for which a client has a stale OHS or none at all. The difference between the "Q/U" and "Un-cached OHS" lines is the additional round trip required to retrieve a current OHS. Even if the OHS must be read before performing an update, the Q/U protocol exhibits faults-scalability. For a given service implemented with the Q/U protocol, the usage pattern of objects will dictate whether most updates are based on a current cached OHS or not.

Non-preferred quorums. Accessing an object at a non-preferred quorum requires some servers to object sync. To quantify the benefit that preferred quorums provide, we implemented a random quorum access policy against which to compare. We measured the average response time for a single client process performing **increment** operations on a single counter object. For the case of $b = 1$ with $n = 6$, the response time using the preferred quorum access policy is 402 µs. Using the random quorum access policy, the average client response time is 598 µs. Because of the small quorum system size, the random policy issues updates to the quorum with the latest object version one in six times. For the other five in six, it takes a server 235 µs to perform an object sync. In the prototype implementation, to perform such an object sync if $b = 1$, a server retrieves the desired object version from two ($b + 1$) other servers.

We also measured the peak throughput provided as the number of faults tolerated increases. The Random Quorum line in Figure 4.5 shows the results. Comparing the Random Quorum and Q/U lines, it is clear that the locality of access afforded by the preferred quorum access policy provides much efficiency to the Q/U protocol. As the number of faults tolerated increases, the cost of object syncing, given a random quorum access policy, increases for two reasons. First, the average number of servers that object sync increases. Second, the number of servers that must be contacted to object sync increases.

Contention. Clients accessing an object concurrently may lead to contention in the Q/U protocol. We ran a series of experiments to measure the impact of contention on client response time. In each of these experiments, b = 1 and five clients perform operations. In all cases, one client performs **increment** update operations to a shared counter object. In one set of experiments the other four clients perform **fetch** operations, and in another set, the other four clients also perform **increment** operations. Clients retrieve current replica histories before performing an **increment** operation; this emulates many distinct clients without a cached OHS contending for the counter object. Within a set of experiments, the number of contending clients that perform operations on the shared counter is varied from zero to four.

		Contending Clients				
		0	**1**	**2**	**3**	**4**
fetch	**Isolated**	320	331	329	326	-
	Contending	-	348	336	339	361
increment	**Isolated**	693	709	700	692	-
	Contending	-	1210	2690	4930	11400

Table 4.2: Average response time in µs.

The number of isolated clients that perform operations on unshared counters is four minus the number of contending clients. Five clients are used in each experiment so that the load on servers is kept reasonably constant. The results of these experiments are reported in Table 4.2.

The **fetch** results demonstrate the effectiveness of the inline repair and optimistic query execution optimizations: queries do not generate contention with a single concurrent update. The results of the **increment** experiments show the impact of the backoff policy on response time. Our backoff policy starts with a 1000 µs backoff period and it doubles this period after each failed retry. As contention increases, the amount clients backoff, on average, increases. Not shown in Table 4.2, is that as update contention increases, the variance in per client response time also increases. For example, the standard deviation in client response time is 268 µs with one contending increment client and 4130 µs with four contending **increment** clients.

4.4.4 Q/UNFSv3 Metadata Service

To explore a more complete service, we built a metadata service with the Q/U protocol. The Q/U-NFSv3 metadata service exports all namespace/directory operations of NFSv3. Two types of Q/U objects are implemented: *directory* objects and attribute objects. Attribute objects contain the per-file information expected by NFS clients. Directory attributes are stored in the directory object itself rather than in a separate attribute object, since they are needed by almost all exported methods on directories. Directory objects store the names of files and other directories. For each file in a directory, the directory object lists the object ID of the attribute object for the file. The metadata service is intended to only service namespace operations; this model of a metadata service separate from bulk storage is increasingly popular (e.g., both Farsite [Adya02] and Pond/OceanStore [Rhea03] use this approach).

Table 4.3 lists the average response times for the operations that the Q/U-NFSv3 metadata service exports. These results are for a configuration of servers that tolerates a single malevolent server (i.e., b = 1 and n = 6). The operation type (query or update) and the types of objects accessed are listed in the table. The cost of a single-object query, relative to a single-object update, is illustrated by the difference in cost between **getattr** and **setattr**. The cost of a multi-object update, relative to a single object update, is illustrated by the difference in cost between create and **setattr**. Multi-object updates, such as **create** operations, may span objects with different preferred quorums. For the create operation, the multi-object update is sent to the preferred quorum of the directory object. As such, the first access of the created attribute object incurs the cost of an inline repair (if it does not share the directory object's preferred quorum).

Table 4.4 lists results of running the metadata-intensive PostMark benchmark [Katcher97] on a single client for different fault-tolerances (from $b = 0$ to $b = 5$). PostMark is configured for 5000 files and 20000 transactions. Since Q/UNFSv3 only services namespace operations, file contents are stored locally at the client (i.e., the read and append phases of PostMark transactions are serviced locally). The decrease from $b = 0$ to $b = 1$ occurs because PostMark is single-threaded. As such, Q/U-NFSv3 incurs the cost of contacting more servers (and of larger authenticators) but does not benefit from quorum throughput-scalability. As with the counter object micro-benchmarks however, performance decreases gradually as the number of malevolent servers tolerated increases.

Operation	**Type**	**Objects**	**Response Time (μs)**
getattr	query	attribute	570
lookup	query	directory	586
readlink	query	directory	563
readdir	query	directory	586
setattr	update	attribute	624
create	update	attr. & dir.	718
link	update	attr. & dir.	690
unlink	update	attr. & dir.	686
rename	update	2 attr. & 2 dir.	780

Table 4.3: Q/U-NFSv3 metadata service operations.

Faults Tolerated (b)	0	1	2	3	4	5
Transactions/second	129	114	102	93	84	76

Table 4.4: Q/U-NFSv3 PostMark benchmark.

4.5 Conclusions

The Q/U protocol supports the implementation of arbitrary deterministic services that tolerate the Byzantine failures of clients and servers. The Q/U protocol achieves efficiency by a novel integration of techniques including versioning, quorums, optimism, and efficient use of cryptography. Measurements for a prototype service built using our protocol shows significantly better fault-scalability (performance as the number of faults tolerated increases) in comparison to the same service built using a popular replicated state machine implementation. In fact, in contention-free experiments, its performance is better at every number of faults tolerated: it provides 8% greater throughput when tolerating one Byzantine faulty server, and over four times greater throughput when tolerating five Byzantine faulty servers. Our experience using the Q/U protocol to build and experiment with a Byzantine fault-tolerant NFSv3 metadata service confirms that it is useful for creating substantial services.

5 NESTED OBJECTS IN A BYZANTINE QUORUM-REPLICATED SYSTEM

5.1 Overview

This chapter presents the design and implementation of a framework to support Byzantine fault-tolerance [Lamport82] in a distributed, object-based system. In modern object-based systems, it is commonplace that objects are passed as arguments to and can invoke methods on other objects. A goal of our framework is to support these natural models of object interaction seamlessly from the programmer's perspective, while utilizing object replication and Byzantine fault-tolerant method invocation protocols to mask the Byzantine (arbitrary) failure of a limited number of replicas of each object.

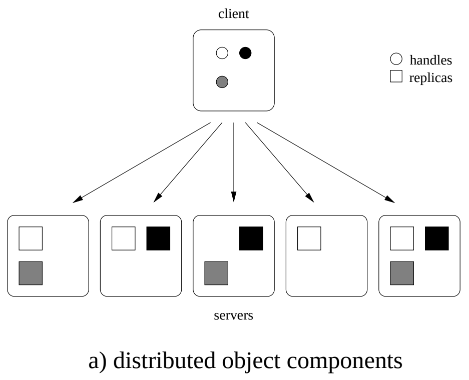
The model of object interaction that our framework supports is motivated by that of Java remote objects. A Java *remote object* is one that can be invoked from outside the *Java Virtual Machine* (JVM) in which it resides, via a protocol called *Remote Method Invocation* (RMI). The client JVM of a remote object holds a proxy for the remote object, called a *stub*, which implements the same interface as the remote object. The client program can invoke methods on the remote object by invoking them on the stub, and can pass the stub as a parameter to other, possibly remote, objects. Those objects that then hold the stub can invoke methods on the remote object, as well. This mechanism thus provides location transparency for calls to the remote object.

In this work, we consider a system we are implementing whereby a serializable Java object can be dynamically exported outside the JVM in which it was created, and replicated to a number of other server JVMs, yielding a *distributed object*. After this operation, the client JVM is left with a *handle* (conceptually similar to a *stub*, but functionally different), again that implements the same interface as the original object; see Figure 5.1(a). Method invocations on the handle are translated to method invocations on a set (*quorum* [Malkhi98a]) of replicas for the distributed object. Like RMI stubs, handles can be passed as parameters to method invocations, potentially on other objects that have been distributed in this way, resulting in object nesting; see Figure 5.1(b). Those JVMs that hold a handle for the distributed object are called *clients*; however, clients of one distributed object can be servers for other distributed objects, so when we refer to a client or a server, it indicates the role in which it is participating at that time.

With nested objects, it is no longer desirable to allow unfettered access to a distributed object by any client that possesses a handle for that object. In particular, we cannot allow a single client replica to perform arbitrary operations on another distributed object: doing so could result in duplicate method invocations when other client replicas perform the same method invocation, and Byzantine-faulty client replicas could corrupt the embedded object, from the application perspective, by invoking incorrect methods on it. Instead, the central goals of our framework are to ensure that (i) only those method invocations endorsed by correct replicas of the calling object are performed, and that (ii) the method invocation protocol itself is robust to a limited number of Byzantine faulty client replicas and server replicas.

For (i), we propose an authorization framework for nested method invocations. This framework authorizes method invocations on distributed objects from single trusted clients as well as from quorums of individually untrusted clients. When object o_1 is passed to o_2, authorization for quorums of o_2's replicas to invoke o_1 is transparently delegated with the use of delegation keys and certificates. For (ii), we develop a new quorum based method invocation protocol that ensures linearizable [Herlihy90] object invocations of method invocations at arbitrary depths, again despite Byzantine failures of a limited number of replicas of each distributed object. This protocol makes no assumptions on message transmission times, i.e. it is designed to function correctly in an asynchronous environment.

We have implemented our framework within a significant restructuring of the Fleet system [Malkhi01]. The initial Fleet system from which we began this implementation did not support nested objects



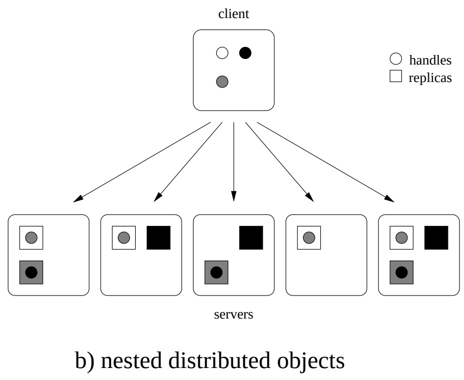


Figure 5.1: Distributed objects.

seamlessly, provided no tolerance for Byzantine client invocations, and did not implement an authorization scheme. As such, our framework is a significant advance in this context.

5.2 Background

To our knowledge, the only prior system to support linearizable access to nested objects in the face of Byzantine faults is Immune [Narasimhan99]. Immune takes a different approach in that it implements every distributed object using *state machine replication* [Schneider90] with majority voting, in which every method invocation is executed by every object replica. Nested method invocations are performed with relative simplicity in the Immune system, as its use of Byzantine fault-tolerant atomic broadcast (specifically [Kihlstrom01]) ensures that all client and server replicas receive every communication in the same order. In contrast, our approach implements quorum-based access: each method invocation involves accessing only a randomly selected quorum of replicas, which can be relatively small, e.g., $O\left(\sqrt{bn}\right)$ replicas for a distributed object with n replicas and tolerating b Byzantine replica failures [Malkhi00b]. While offering improved scalability and load dispersion, the quorum approach introduces challenges not present in the state machine approach, notably the absence of atomic multicast among all client and server replicas to coordinate invocations. Finally, our authorization framework has no analog in Immune, which permits any replicated object to invoke arbitrary methods on another replicated object.

Benign fault-tolerant nesting of transactions has previously been a topic of study in the context of database systems [Moss81]. Nested transactions achieve concurrency atomicity in the form of serializability [Moss86]. While necessary for transactional systems that must apply multiple operations on potentially many objects, serializability is both *non-local* and *blocking*, thus requiring global coordination and locking to enforce it. Serializability was recently studied in the context of JavaSpace transactions [Busi01]; however, while there can be nested transactions in JavaSpaces, there is no analog to our nested method invocations as JavaSpaces contain passive objects that can only be read and written, rather than remote objects on which methods can be invoked. Linearizability, the form of concurrency atomicity that we pursue in this work, provides concurrency atomicity of single operations performed on a single object only [Herlihy90].

Though weaker than serializability, we have opted for linearizability for two reasons. First, returning to the object sharing model that motivates our work, linearizability is the concurrency atomicity property that is achieved by Java RMI (though not in a fault-tolerant way), provided that the remote object processes requests sequentially. As such, our system will be suited to applications already utilizing that model. Second, linearizability can be enforced locally, and avoids locking, which is problematic when a node responsible for unlocking an object fails. Nevertheless, we intend to explore serializability in future work.

5.3 Authorization Framework

As discussed in Section 5.1, object nesting in combination with Byzantine failures requires that we depart from a model in which simply possessing a handle for a distributed object is sufficient to invoke methods on it. Otherwise, a faulty object replica which was given a handle for another object through servicing a method invocation would then be able to invoke arbitrary methods on that object. Our goal is to allow the creator of a distributed object to invoke methods on that object, and to delegate that authority to another client object, which itself may be replicated in order to withstand the Byzantine failure of some of its replicas. We anticipate that this delegation will most commonly occur automatically when a handle to one distributed object is passed as a parameter to a method invocation on a second distributed object.

5.3.1 Assumptions

As discussed previously, the environment that we consider executes methods on a distributed object at a *quorum* of its replicas, and the set of allowable quorums constitutes a *quorum system* for the distributed object. For the purposes of the present section, we are not concerned with the structure of the quorum system, except for one assumption: the quorum system is formed based on an assumed maximum number b of its replicas that will suffer Byzantine failures. For example, it is necessary that each quorum be larger than b, lest operations be performed at a quorum of only faulty servers, and it is also necessary that any b failures leaves a quorum available, so that operations can be completed. Such quorum constructions can be found in, e.g., [Malkhi98a, Malkhi00b].

We assume that communications to and from servers are protected using standard cryptographic techniques. At a high level, our strategy will be to permit only a number of replicas of size $b_1 + 1$ or greater of a distributed object o_1 to invoke methods on another distributed object o_2. Here, b_1 denotes the number of replica failures that the quorum system for o_1 was designed to survive. In this way, any invocation by any replica of o_1 that the correct replicas of o_2 accept is corroborated by a correct replica of o_1. For the purposes of this section, we treat trusted individual clients that are permitted to invoke methods on a distributed object, e.g., the creator of the object or another client to which it explicitly passes the handle through an out-of-band mechanism, as a special case of a distributed object o_1 with $n_1 = 1$ replicas and $b_1 = 0$ faults.

5.3.2 Delegation

The starting point for method invocation authorization is the principal that originally creates a distributed object, i.e., the client that exports the object from its JVM to make a new distributed object. When the distributed object is created, its creator generates a new private digital signing key S and corresponding public verification key V (e.g., [Rivest78, Kravitz93, ANSI99]) and deploys V with each replica, as the "root" key for the distributed object. The creator can optionally deploy additional root public keys with each replica, though here we restrict our attention to a single root key.

The correct replicas of this distributed object will henceforth only permit method invocations bearing digital signatures that can be verified with V, or for which the root key has delegated authority (perhaps transitively). This delegation can occur in two ways. The most straightforward is explicit delegation by the application, in which the object creator certifies a public key provided by another potential client as being authorized to access the distributed object. This form of delegation closely follows that of, e.g., Gasser and McDermott [Gasser90], and will not be detailed here. The second and more complex form of delegation occurs implicitly, when objects are nested. To support this form of delegation, each handle contains a private signature key called the *handle key*—the handle key for the initial handle is the private root key of the handle—plus a set of *statements* (certificates) regarding the keys for which that handle key bears authority. To fully describe how delegation works, we need to consider two cases: one in which a handle for one distributed object is passed as a parameter into a method call on another distributed object, and one in which a method call on one distributed object returns a handle of another distributed object.

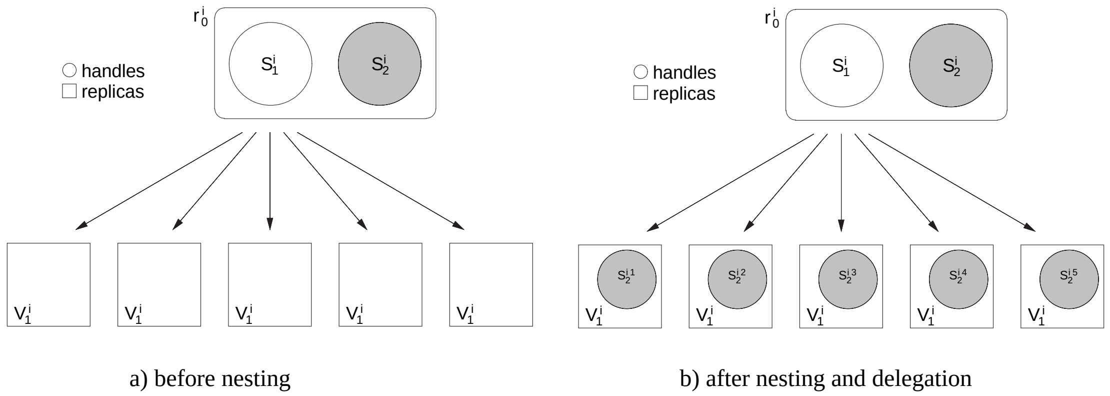


Figure 5.2: Nested objects.

Consider an object o_0 with n_0 replicas, each replica r_0^i of which holds handles h_1^i and h_2^i for distributed objects o_1 and o_2, respectively. Figure 5.2(a) shows an instance in which $n_0 = 1$, as would be the case if o_0 were a non-replicated client. Let S_2^i denote the private handle key for h_2^i. If r_0^i passes h_2^i in a method call parameter to h_1^i, then h_2^i makes a copy h_2^{ij} of itself for replica r_1^j (i.e., the j-th replica of o_1), except h_2^{ij} is equipped with a newly generated public/private key pair (V_2^{ij}, S_2^{ij}) before being sent to r_1^j. The results of this delegation in the case of Figure 5.2(a) are shown in Figure 5.2(b).

After generating a new public/private key pair for each replica of o_1, a new statement must be constructed authorizing a non-faulty set of o_1's replicas to invoke methods on o_2. Let $\mathcal{V}_2^i$ denote the set $\{V_2^{i1}, \ldots, V_2^{in_1}\}$ of public keys created for o_1's replicas, where n_1 is the number of replicas of o_1, and let b_1 be the number of faults the quorum construction of o_1 is designed to mask. Then for each replica r_1^i of o_1, the statement set of h_2^{ij} is augmented with a statement of the form

$$V_2^i \text{ says } (b_1 + 1 \text{ of } \mathcal{V}_2^i) \Rightarrow V_2^i, \qquad (5.1)$$

i.e., a certificate signed by S_2^i stating that any subset of $b_1 + 1$ keys in $\mathcal{V}_2^i$ is authorized with the same privileges as V_2^i. ("Says" and "speaks for" ($\Rightarrow$) are common formalisms for expressing credentials, e.g., [Lampson92, Ellison99, Appel99, Balfanz00, Bauer02].) Verifying a "signature" σ on a message m with ($b_1 + 1$ of $\mathcal{V}_2^i$) means verifying that at least $b_1 + 1$ of the public keys in $\{V_2^{i1}, \ldots, V_2^{in_1}\}$ can be used to verify signatures in σ (a set) for m.

Finally, once at least $b_0 + 1$ of handles $h_2^{1j}, h_2^{2j}, \ldots, h_2^{n_0 j}$ have been deployed at r_1^j, r_1^j can coalesce these handles into a single handle $\hat{h}_2^j$ by generating a new public/private key pair ($\hat{V}_2^j$, $\hat{S}_2^j$) and the certificate[3]

[3] The handles $h_2^{1j}, h_2^{2j}, \ldots, h_2^{n_0 j}$ need not all be deployed to r_1^j, and some may not be due to failures. Thus, at any point in time this certificate will concern only the keys S_2^{ij} that r_1^j has received so far, and can be updated when another handle is received.

$$\left(\underset{i}{AND}\ V_2^{ij}\right) \text{ says } \left(\hat{V}_2^j \Rightarrow \underset{i}{AND}\ V_2^{ij}\right) \tag{5.2}$$

In this way, requests issued by $\hat{h}_2^j$ to o_2 replicas will need only sign with $\hat{S}_2^j$, versus each of S_2^{1j}, . . . , $S_2^{n_0 j}$. Of course, the statement set of $\hat{h}_2^j$ contains both (5.2) and the union of the statement sets of h_2^{1j}, . . . , $h_2^{n_0 j}$, which includes (5.1).

Safety. Consider a method invocation *m* executed by only faulty handles $\{\hat{h}_2^j\}$, i.e., by at most b_1 replicas $\{r_1^j\}$. Each such invocation is signed by $\hat{S}_2^j$, and so each replica r_2^k that sees this invocation can determine that $\hat{V}_2^j$, says *m* and thus that

$$\left(\underset{i}{AND}\ V_2^{ij}\right) \text{ says } m$$

by (2). However, since there are at most b_1 *j*'s for which this holds true, it is not possible to infer that (b_1 + 1 of $\mathcal{V}_2^i$) says *m*, or thus that V_2^i says *m* for any correct r_0^i. Consequently, if any invocation *m′* to o_2 by o_0 requires b_0 + 1 replicas of o_0 to submit *m′*, then any invocation *m* to o_2 by o_1 will not succeed if only b_1 replicas of o_1 submit *m*, and safety follows by induction.

Cost. The latency of the above delegation is dominated by the costs of (i) generating the n_1 private key pairs (S_2^{i1}, V_2^{i1}), . . . , ($S_2^{in_1}$, $V_2^{in_1}$) at r_0^i; (ii) generating the private key pair ($\hat{S}_2^j$, $\hat{V}_2^j$) at r_1^j; and (iii) generating the digital signatures for (1) and (2) at r_0^i and r_1^j, respectively. For digital signature schemes for which key generation is costly, notably RSA [Rivest78], it would be necessary to pre-generate these keys in the background and store them for use when needed. For this reason, it would be preferable to use a digital signature scheme, such as DSA [Kravitz93] or ECDSA [ANSI99], for which key generation is very efficient [Wiener98].[4]

A method invocation following this delegation, i.e., in which object o_1 invokes a method on object o_2, requires each invoking replica r_1^j to digitally sign its request with $\hat{S}_2^j$. Replica r_2^k, upon receiving such a request, must verify not only its signature but also the signatures of the statement sets forwarded with the request. This cost is particularly important since as nesting depth increases, the size of generated statement sets grows. While r_2^k will incur this cost on the first method invocation, caching the verification status of statements (certificates) should significantly decrease the cost of subsequent method invocations. Nonetheless, nested method invocations performed after one object has been nested in another will be dominated by the cost of signature verification, and would thus benefit from a digital signature scheme, such as RSA, where verification was very efficient [Wiener98]. As keys can be pre-generated, but not

[4] Key generation in DSA is efficient once certain global parameters are fixed, i.e., primes typically denoted by *p* and *q*, and a generator *g* of a subgroup of order *q* in the integers modulo *p*. Similarly, ECDSA is typically defined over a fixed curve, which then allows efficient key generation.

pre-verified, the cost of signature verification is the determining factor in selecting a signature algorithm for use in a practical system.

Returning Handles. Our framework accommodates returning a handle from a method invocation on a distributed object in a very similar way. For this, we continue from the above example, and consider that after the above has transpired, replicas of o_0 invoke a method on their corresponding handles for o_1, and in doing so should obtain handles for o_2 that should permit the o_0 replicas to invoke methods on o_2 in the future. (o_0 need not be the same object as in the preceding example, though to simplify notation we consider it to be.) Note that for r_1^j to perform the invocation, it must be invoked by $b_0 + 1$ replicas of o_0, and it will return a copy $\hat{h}_2^{ij}$ of its handle $\hat{h}_2^j$ to the handle h_1^i in r_0^i. Again, however, r_1^j will replace $\hat{S}_2^j$ in $\hat{h}_2^{ij}$ with a newly generated handle key $\hat{S}_2^{ij}$, and add

$$\hat{V}_2^j \text{ says } (b_0 + 1 \text{ of } \hat{\mathcal{V}}_2^j) \Rightarrow \hat{V}_2^j \tag{5.3}$$

to the statement set of $\hat{h}_2^{ij}$, where $\hat{\mathcal{V}}_2^j = \{\hat{V}_2^{1j}, \ldots, \hat{V}_2^{n_0 j}\}$ and b_0 is the number of failures the quorum system for o_0 is designed to mask. Just as r_1^j did in the previous case, when handle h_1^i (in replica r_0^i) receives at least $b_1 + 1$ handles $\hat{h}_2^{ij}$, it coalesces these handles to build a handle $\bar{h}_2^i$ with a new handle key $\bar{S}_2^i$ and a statement set including

$$\left(\underset{j}{AND}\ \hat{V}_2^{ij}\right) \text{ says } \left(\bar{V}_2^i \Rightarrow \underset{j}{AND}\ \hat{V}_2^{ij}\right) \tag{5.4}$$

h_1^i then returns $\bar{h}_2^i$ from the method invocation, to r_0^i. As before, if only b_0 replicas of o_0 now attempt to perform a method invocation m on o_2, then no correct replica of o_2 will infer ($b_0 + 1$ of $\hat{\mathcal{V}}_2^j$) says m, and so method m will not be invoked. The cost analysis of this case is similar to that above.

Revocation. Distributed objects are typically intended to be long-lasting, and in particular, to outlive the clients that create them. As a result, it is not practical for the certificates created during delegation (i.e., (1)–(4)) to expire; doing so would leave distributed objects stranded with no clients able to access them. As a result, we opt for a different form of revocation, conceptually similar to the approach in Gasser and McDermott [Gasser90]: when a JVM no longer requires a reference to a handle and so the handle is garbage collected, its private handle key is deleted. This implies that once the correct replicas of a distributed object o_1 have deleted their handles for another distributed object o_2, the delegation that permits o_1 to invoke methods on o_2 can no longer be exercised.

5.4 Operation Ordering

As discussed in Section 5.1, the model of object interaction that we attempt to mimic in our approach is that offered by Java RMI. The concurrency semantics that most naturally characterize Java RMI, presuming that remote Java objects process each method invocation in isolation, is linearizability [Herlihy90]. For the distributed objects we consider, owing to their replication and quorum based access, we require a method invocation protocol that implements this property. Prior protocols to achieve this property in systems supporting quorum-based access, notably [Chockler01], assume a correct client. This assumption is violated in nested object systems, where clients can be replicas of other distributed objects, some of which may be Byzantine faulty.

The approach we take to method invocations here retains the basic structure of prior quorum-based protocols in permitting a single client to drive the ordering protocol, but does so without trusting that client to make any protocol decisions. Rather, this client is used only as a point of centralization to distribute a set of server messages from which the servers work to make protocol decisions. Forms of misbehavior, notably sending different messages to different servers, cannot cause correct servers to take permanent and conflicting actions. In addition, a faulty client cannot prevent progress by falling silent as another client can unilaterally "take over" driving the protocol.

In order to ensure linearizability, all correct servers must apply operations in the same order. They do this by indirectly communicating state with other server replicas in a quorum, and then making deterministic decisions based upon the view they have of the quorum. When a non-faulty client drives the protocol, it echoes the state of each server replica in a quorum to the other server replicas in some quorum. Server state consists of the set of pending operations on that replica, and the last distributed object state committed (σ^c), proposed (σ^{pc}), and suggested (σ^s) on that replica.

5.4.1 Client Protocol

The client side of the method invocation protocol is shown in Figure 5.3. This protocol consists of two main parts, shown in lines 2–10 and 11–23 of Figure 5.3, run concurrently as indicated by the "||" in lines 2 and 11. In the first part, the client request ("*op*") is forwarded to a quorum Q drawn deterministically ("$\leftarrow_D$", line 4) from a quorum system $\mathcal{Q}$. Deterministic quorum selection ensures that each server replica in the selected quorum will be contacted by at least $b_c + 1$ client replicas. The client signs its request ("$S(op)$", line 6) using its private key (the handle key of Section 3) and sends the request to each member of Q. It collects responses ρ into a set R_1 until it has at least $b_s + 1$ replicas (line 8), where b_s is the number of replica failures that the quorum system $\mathcal{Q}$ is designed to withstand.

In parallel, the client drives the ordering protocol as shown in lines 11–23 of Figure 5.3. The client runs this ordering protocol with a particular *rank r*, which is an integer value assigned anew by the `chooseRank` method each time the client begins the ordering protocol (line 13). In order to make progress, the same rank must be kept during each iteration, so the standard behavior of `chooseRank` is to leave the current rank unchanged. However, at any point in time each server will only interact with the highest ranked client that has contacted it so far. As such, when a client receives a `RankException` from a server (not shown in Figure 5.3), `chooseRank` selects a new, larger rank for use in the next round of the ordering protocol. A backoff strategy could be used to avoid clients repeatedly interrupting each other; as this does not affect the underlying protocol, we discuss it in Section 5.4.3.

```
1. submit($op$)
2.        ||   $waiting \leftarrow$ true
3.             $R_1 \leftarrow \emptyset$
4.             $Q_1 \leftarrow_D \mathcal{Q}$
5.             $||_{u \in Q1}$
6.                  $\rho_u \leftarrow u$.submit($S(op)$)
7.                  $R_1 \leftarrow \rho_u \cup R_1$
8.             **until** ($\exists \rho : |\{\rho_u \in R_1 : \rho = \rho_u\}| \geq b_s + 1$)
9.             $waiting \leftarrow$ false
10.            **return** $\rho : |\{\rho_u \in R_1 : \rho = \rho_u\}| \geq b_s + 1$
11.       ||   $R_2 \leftarrow \emptyset$
12.            **repeat**
13.                 $r \leftarrow$ chooseRank()
14.                 $R_3 \leftarrow \emptyset$
15.                 $Q_2 \leftarrow \mathcal{Q}$
16.                 $||_{u \in Q2}$
17.                      state$_u \leftarrow u$.process($R_2$, $r$)
18.                      **if** (valid( state$_u$,$R_2$, $r$,$V_u$))
19.                           $Q_3 \leftarrow \{u\} \cup Q_3$
20.                           $R_3 \leftarrow$ state$_u \cup R_3$
21.                 **until** ($\exists Q \in \mathcal{Q} : Q \subseteq Q_3$)
22.                 $R_2 \leftarrow R_3$
23.            **until** (waiting = false)
```

Figure 5.3. Client side of ordering protocol.

The core of this part of the client protocol is that it simply queries each server u in a quorum $Q_2 \in \mathcal{Q}$ (lines 15-16) by querying u.`process`(R_2, r) (line 17) where R_2 is the set of valid responses gathered from servers in the previous iteration, until it receives valid responses from a quorum $Q \in \mathcal{Q}$ (line 21). Here, a response state is *valid* if the state is consistent with the correct execution of *u*.process(*R*2, *r*), which includes bearing the current rank *r*, being properly signed by S_u, and bearing values reflecting correct protocol logic when applied to R_2. Validity is tested in line 18, though this is not shown in the figure to avoid duplicating the server-side logic. Although validity testing by the client can not be relied on to ensure correctness (as the client could be faulty), it is important in allowing correct clients to determine when valid responses have been received from a full quorum, without which the next round may be futile. Note that a valid response is required from every server in a full quorum, which implies that a new quorum may need to be selected if any members of the original quorum don't cooperate; alternatively non-blocking quorums [Bazzi01] could be used, effectively contacting extra servers with the guarantee that correct responses will be received from a full quorum.

A malicious client could fail to send messages to some or all of the server replicas or at the very worst could send different messages to different sets of server replicas. As we will demonstrate shortly, the only possible effect of such misbehavior is that it might temporarily slow progress, but this cannot result in an incorrect linearization of method invocations.

5.4.2 Server Protocol

The server side of the protocol, as outlined in Figure 5.4, is necessarily more complicated as it handles all of the protocol's logic. The `process` method (on the left of Figure 5.4) is invoked by client replicas' ordering threads, while the remaining methods (on the right of Figure 5.4) are intended strictly for internal use by a server replica. For simplicity we do not show the `submit` method invoked by clients' submit threads; this simply adds the submitted operation to the *pending* set once it has been requested by b_c+1 client replicas, and returns a response to the waiting client replicas when the submitted operation has been ordered and executed.

The bulk of the server-side logic is contained in the process method. This method first confirms that the rank of the client is at least as large as any rank seen so far, and throws a `RankException` and terminates this run of process if not (lines 2–6). After checking the rank and verifying the signatures of the supplied server replica states (line 9), each server replica takes steps to determine which new object state should be *suggested*, *proposed* or *committed* next. It does this on the basis of the following values, which it derives from the server replica states $\{\text{state}_u\}_{u \in Q}$.

- σ_g^s (line 13) is the state last suggested by some quorum of servers, if any;
- σ_g^{pc} (line 17) is the state with the highest version number proposed to some correct server or, if there are multiple such states, the one proposed by the highest ranked proposer;
- σ_g^c (line 21) is the state with the highest version number committed to some correct server; and
- *completed* is the highest version number for which some quorum has committed a state with that version number (line 24), or for which some correct server has suggested (line 14) or proposed (line 19) a state with a strictly higher version number. We say that a state has completed if its version number is less than or equal to *completed* at some correct server.

```
1.  process({ $\text{state}_u$ }$_{u \in Q}$, $r$)
2.      if ($r$ > maxRank)
3.          maxRank ← $r$
4.          $\sigma^s \leftarrow \perp$
5.      else if ($r$ < maxRank)
6.          throw RankException(maxRank)

7.      foreach $u \in Q$
8.          $\langle \text{content}_u, r_u \rangle \leftarrow \text{state}_u$
9.          if ($r_u = r \wedge V_u$ says $\text{state}_u$)
10.             $\langle \sigma_u^c, \sigma_u^{pc}, \sigma_u^s, \text{pending}_u \rangle \leftarrow \text{content}_u$
11.         else $Q \leftarrow Q \setminus u$

12.     if ($\exists\, Q' \in \mathcal{Q} : Q' \subseteq Q$)
13.         $\sigma_g^s \leftarrow \sigma : \exists\, Q' \in \mathcal{Q} : Q' \; \{u : \sigma = \sigma_u^s\}$
14.         completed ← $\max\{v : |\{u : \sigma_u^s.\text{version} > v\}| \geq b_s + 1\}$

15.         $\Sigma_g^{pc} \leftarrow \{\sigma : |\{u : \sigma = \sigma_u^{pc}\}| \geq b_s + 1\}$
16.         $\Sigma_g^{pc'} \leftarrow \{\sigma \in \Sigma_g^{pc} : \sigma.\text{version} = \max_{\sigma' \in \Sigma_g^{pc}} \{\sigma'.\text{version}\}\}$
17.         $\sigma_g^{pc} \leftarrow \sigma \in \Sigma_g^{pc'} : \sigma.\text{proposer} = \max_{\sigma' \in \Sigma_g^{pc'}} \{\sigma'.\text{proposer}\}$
18.         $Q_{pc} \leftarrow \{u : \sigma_u^{pc} = \sigma_g^{pc} \wedge \sigma_u^{pc}.\text{proposer} = r\}$
19.         completed ← max{completed,
                $\max\{v : |\{u : \sigma_u^{pc}.\text{version} > v\}| \geq b_s + 1\}\}$

20.         $\Sigma_g^c \leftarrow \{\sigma : |\{u : \sigma = \sigma_u^c\}| \geq b_s + 1\}$
21.         $\sigma_g^c \leftarrow \sigma \in \Sigma_g^c : \sigma.\text{version} = \max_{\sigma' \in \Sigma_g^c} \{\sigma'.\text{version}\}$
22.         $Q_c \leftarrow \{u : \sigma_u^c = \sigma_g^c\}$
23.         if ($\exists\, Q' \in \mathcal{Q} : Q' \subseteq Q_c$)
24.             completed ← max{completed, $\sigma_g^c$.version}

25.         if ($\sigma_g^c \neq \perp \wedge \sigma_g^c$.version > completed)
26.             commit($\sigma_g^c$)
27.         else if ($\sigma_g^{pc} \neq \perp \wedge \sigma_g^{pc}$.version > completed)
28.             if ($\exists\, Q' \in \mathcal{Q} : Q' \subseteq Q_{pc}$)
29.                 commit($\sigma_g^{pc}$)
30.             else if ($\sigma_g^{pc} = \sigma_g^s$)
31.                 propose($\sigma_g^{pc}$, $r$)
32.             else
33.                 suggest($\sigma_g^{pc}$)
34.         else if ($\sigma_g^s \neq \perp \wedge \sigma_g^s$.version > completed)
35.             propose($\sigma_g^s$, $r$)
36.         else
37.             $\textit{pending}_g \leftarrow \{op : |\{u : op \in \textit{pending}_u\}| \geq b_s + 1\}$
38.             if ( $\textit{pending}_g \neq \emptyset$)
39.                 $\sigma \leftarrow$ addOps($\sigma_g^c$, $\textit{pending}_g$)
40.                 suggest($\sigma$)

41. return $S(\langle \sigma^c, \sigma^{pc}, \sigma^s, \textit{pending} \rangle, r$
```

```
1.  addOps($\sigma'$, $\textit{pending}$)
2.      repeat
3.          $op \leftarrow_D \textit{pending}$
4.          $\textit{pending} \leftarrow \textit{pending} \setminus \{op\}$
5.          if ($\sigma'$.reflects($op$) = false)
6.              $\sigma'$.addOp($op$)
7.      until ($\textit{pending} = \emptyset$)
8.      $\sigma'$.version ← $\sigma'$.version + 1
9.      return $\sigma'$
1.  suggest($\sigma$)
2.      if ($\sigma^s = \perp$)
3.          $\sigma^s \leftarrow \sigma$
4.      else
5.          throw RankException($\textit{maxRank}$)

1.  propose($\sigma$, $r$)
2.      $\sigma$.proposer ← $r$
3.      $\sigma^{pc} \leftarrow \sigma$

1.  commit($\sigma$)
2.      if ($\sigma$.version > $\sigma^c$.version)
3.          $\sigma$.applyOps
4.          $\sigma^c \leftarrow \sigma$
5.          $\textit{pending} \leftarrow \textit{pending} \setminus$
                $\{op : \sigma.\text{reflects}(op) = \text{true}\}$
6.          $\textit{response} \leftarrow \textit{response} | \sigma.\ \textit{response}$
```

Figure 5.4: Server side of ordering protocol.

In summary, the server takes the following actions, in order. It commits σ_g^c if it cannot determine that it has been committed at a full quorum (lines 25–26). Otherwise, it looks at σ_g^{pc} if it exists and is not already completed (line 27): if σ_g^{pc} has been proposed to a full quorum by this client (lines 18, 28), it commits σ_g^{pc} (line 29). If σ_g^{pc} has not been proposed to a full quorum by this client, the server suggests σ_g^{pc} if it has not already been suggested at a full quorum (line 33) and proposes σ_g^{pc} if it has (line 31). If σ_g^{pc} does not exist or is already completed, then the server similarly examines σ_g^{s}, proposing it if it exists and is not already completed (lines 34–35). If σ_g^{s} does not need to be acted upon, the server deterministically orders method invocations that are pending on $b_s + 1$ servers—appending them to the current state, but not yet executing them; see below—through the `addOps` operation, and suggests this state (lines 37–40). We note that in `addOps`, the predicate σ.`reflects`(*op*) indicates whether or not the operation *op* is incorporated into the state σ; as such, it prevents duplicate invocations. Finally, the server signs its current local states that it sees as committed, proposed, and suggested, as well as all pending invocations and the rank *r*, and returns this (line 41). It is important to note that the `addOps` operation (line 39) does not actually invoke method invocations on the object σ_g^{c}; rather, they are just appended for later application by `applyOps` (line 3 of commit). With the introduction of nested operations, the scope of impact of a method invocation on an object is no longer limited to that object. Each invocation can result in a nested method invocation on another object. As such, it is important to withhold performing operations until they are actually committed, as a suggested or even a proposed state may never ultimately be committed.

5.4.3 Liveness

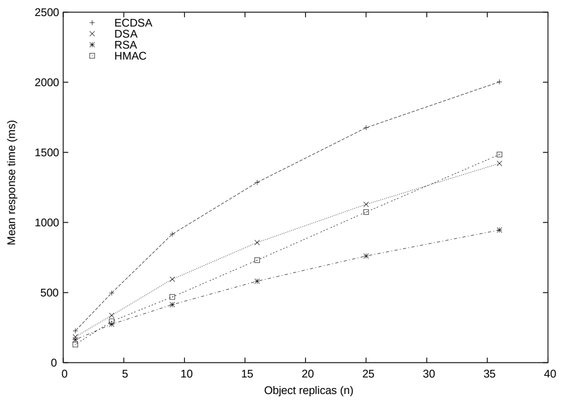
Our discussion so far has focused on safety. Liveness is also an important consideration, especially as a Byzantine client could end up driving the protocol. It is thus desirable to tolerate faulty clients while still achieving high throughput with many concurrent clients. A first step towards this is to require clients to sign their ordering requests with the private key of the associated handle. In conjunction with the delegation statements defined in Section 5.3, this would ensure that the ordering protocol for a given distributed object could only be driven by handles which were authorized to invoke methods on that object.

In addition to ensuring that only authorized handles can drive the ordering protocol, it is also necessary to prevent Byzantine clients from doing all the driving and thus preventing forward progress. We do this with a form of backoff similar to [Chockler01], though enforced by server replicas to prevent faulty clients from repeatedly interrupting the protocol for correct clients. While a faulty client can delay the ordering protocol during a single round, enforced backoff ensures that correct clients (who are the majority due to quorum overlap requirements) will regularly be allowed to drive the protocol. Because operations are applied en masse, high average throughput is maintained by the system despite small delays during rounds which are driven by Byzantine clients.

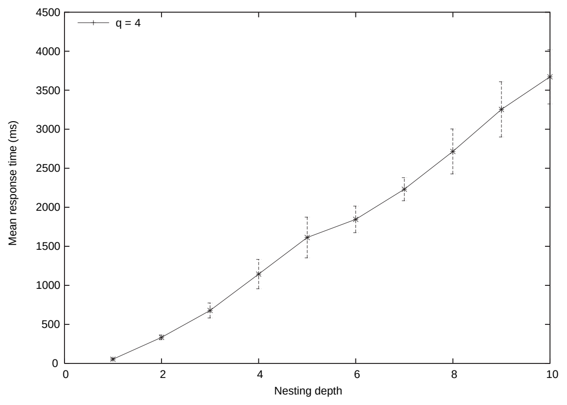
5.4.4 Performance

We have implemented the protocol discussed in Sections 5.4.1 and 5.4.2 within the context of a substantial revision of the Fleet system. As described in Section 1.3, Fleet is a distributed object store built in Java that implements distributed objects such as those shown in Figure 5.1(a). However, Fleet previously did not provide support for transparent nesting; in fact, nesting could result in duplicate nested method invocations, even in the absence of failures [Malkhi01]. As such, our framework provides a useful extension to the Fleet system.

In our experiments, each server and client was equipped with dual Pentium-III 1GHz processors, running Linux 2.4.24 SMP and Java HotSpot™ Server VM 1.4.2. The servers and client were connected by 100Mbps Ethernet and utilized TCP for all communication; multicast was not used. Key generation,



(a) Response time per quorum size, with nesting depth of 1



(b) Response time per nesting depth, with quorum size of 4

Figure 5.5. Mean response time.

signing, and signature verification were all performed natively using the Crypto++ Library. All tests were performed with three signature algorithms: RSA [Rivest78] with 1024-bit keys, DSA [Kravitz93] with 1024-bit keys and ECDSA [ANSI99] with 160-bit keys. By way of comparison, we also adapted our protocol to use HMACs [Bellare96], using Diffie-Hellman key agreement to establish a shared secret key between each pair of servers. The quorum system in use was a minimally-sized threshold quorum system [Malkhi98a], and all method invocations were performed by unreplicated clients.[5] In order to measure the overhead introduced by the ordering protocol, all method invocations performed inexpensive tasks, such as get and set. Due to the limited set of homogeneous computers at our disposal, we only tested distributed objects having between $n = 1$ and $n = 36$ replicas and with $b = 0$, yielding quorums of sizes from $q = 1$ to $q = 20$.

Preliminary response times as seen by a client for a relatively unoptimized implementation of this protocol are shown in Figure 5.5(a). Among the three digital signature algorithms tested here, the use of RSA signatures, whose verification times are notably faster than the other signature algorithms [Wiener98], yielded the lowest mean response time. This result reflects the fact that each server replica in a quorum must sign its own state, but verify signatures on states from every server in a quorum. Note that the response time growth rate is sublinear while b stays constant, due to the fact that quorums in the chosen quorum system are asymptotically of size $O(\sqrt{bn})$. Mean response time as a function of nesting depth is shown in Figure 5.5(b). Single level method invocations from a client to a quorum of replicas executes in 50 ms. Each additional level of nesting, from one quorum of replicas to another, adds about 90 ms to the total method invocation time. The increased cost of invoking methods between quorums, as opposed to from an unreplicated client to a quorum, stems from the fact that authorization to invoke the method must be granted to a quorum instead of a single client, and the results must be authenticated to a quorum instead of a single client. Note that each level of nesting adds a constant amount to the total method invocation cost, keeping the total latency linear with respect to nesting depth.

Prior to analyzing the performance of HMACs, it is necessary to understand how their implementation differs from that of signatures. As part of the ordering protocol, each server replica signs its local state and sends it to the driving client replica (Figure 5.4, line 41), who then distributes it to a quorum of server replicas (Figure 5.3, lines 16–17). When using HMACs instead of signatures, it is necessary for each server replica to include an HMAC for every other server replica, and for the driving client replica to

[5] While replicated clients are an important component of our work, the ordering protocol is driven by a single client replica at any given time.

include the appropriate set of HMACs for each server replica that it contacts. HMACs thus consume more network bandwidth between server replicas and the driving client replica, as the number of server replicas grows. Considering this, it is unsurprising that the response time using HMACs does not scale as well as signatures as *n* increases.

Another difference resulting from HMACs is that the client is unable to verify them (unlike signatures). As such, a faulty server that returns invalid HMACs will not be readily detectable to a client, and may impinge on the client's ability to complete the protocol. In such circumstances, the client can "fall back" to a signature-based protocol to better enforce progress.

5.5 Summary

The delegation architecture which we have presented can be used to arbitrarily nest distributed objects, while ensuring correct system behavior despite a limited number of Byzantine failures in the object replicas. This is accomplished through the creation of delegation keys and delegation certificates that work together to authorize the invocation of methods on nested objects. Method invocation capabilities on a specific distributed object can be delegated to other clients explicitly, and are implicitly delegated to permit nested method invocations by sufficiently large groups of replicas to succeed transparently.

We presented a novel protocol to achieve linearizability of nested method invocations. Our protocol tolerates Byzantine faulty clients, as is necessary when methods are invoked from other distributed objects that themselves must withstand Byzantine faults. We detailed this protocol, argued its correctness, and described its implementation and performance in a distributed object system. An extension that we are presently studying is the nested *creation* of distributed objects, i.e., where one distributed object creates another. Our framework provides a basis for supporting distributed object creation, but requires us to address additional issues.

6 REFERENCES

[Abd-El-Malek05a] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, Jay J. Wylie. Lazy Verification in Fault-Tolerant Distributed Storage Systems. 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005), October 26-28, 2005, Orlando, Florida.

[Abd-El-Malek05b] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. The Read/Conditional-Write and Query/Update protocols. Technical report CMU-PDL-05-107. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, September 2005.

[Abd-El-Malek05c] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, Jay J. Wylie. Fault-Scalable Byzantine Fault-Tolerant Services. SOSP'05, October 23-26, 2005, Brighton, United Kingdom.

[Adya02] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. Symposium on Operating Systems Design and Implementation, pages 1-15. USENIX Association, 2002.

[Aguilera00] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. Distributed Computing, 13(2):99-125. Springer-Verlag, 2000.

[ANSI99] ANSI X9.62, Public Key Cryptography for The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), American National Standards Institute, 1999.

[Appel99] A. W. Appel and E. W. Felten. Proof-Carrying Authentication. In Proceedings of the 6th ACM Conference on Computer and Communications Security, November 1999.

[Baker91] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. ACM Symposium on Operating System Principles. Published as Operating Systems Review, 25(5):198–212, 13– 16 October 1991.

[Balfanz00] D. Balfanz, D. Dean, and M. Spreitzer. A security infrastructure for distributed Java applications. In Proceedings of 2000 IEEE Symposium on Security and Privacy, May 2000.

[Bauer02] L. Bauer, M. A. Schneider, and E. W. Felten. A General and Flexible Access-Control System for the Web. In Proceedings of the 11th USENIX Security Symposium, August 2002.

[Bazzi01] R. A. Bazzi. Access cost for asynchronous Byzantine quorum systems. Distributed Computing 14(1):41–48, 2001.

[Bazzi04] R. A. Bazzi and Y. Ding. Non-skipping timestamps for Byzantine data storage systems. DISC, 2004.

[Becker96] B. Becker, S. Gschwind, T. Ohler, P. Widmayer, and B. Seeger. An asymptotically optimal multiversion btree. VLDB Journal, 5(4):264–275, 1996.

[Bellare96] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. Advances in Cryptology - CRYPTO, pages 1–15. Springer-Verlag, 1996.

[Bernstein87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. Addison-Wesley, Reading, Massachusetts, 1987.

[Blackwell95] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. USENIX Annual Technical Conference, pages 277– 288. USENIX Association, 1995.

[Bracha85] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. Journal of the ACM, 32(4):824-840. ACM, October 1985.

[Busi01] N. Busi and G. Zavattaro. On the Serializability of Transactions in JavaSpaces. In Proc. of International Workshop on Concurrency and Coordination, Electronic Notes in Theoretical Computer Science, Vol. 54, July 2001.

[Cachin01] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In Advances in Cryptology – CRYPTO '01, pages 524–541, August 2001.

[Cachin02] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. International Conference on Dependable Systems and Networks, pages 167-176. IEEE, 2002.

[Cachin05a] C. Cachin and S. Tessaro. Asynchronous verifiable infromation dispersal. Symposium on Reliable Distributed Systems. IEEE, 2005.

[Cachin05b] C. Cachin and S. Tessaro. Brief announcement: Optimal resilience for erasure-coded Byzantine distributed storage. International Symposium on Distributed Computing. Springer, 2005.

[Castro] M. Castro and R. Rodrigues. BFT library implementation. http://www.pmg.lcs.mit.edu/bft/#sw.

[Castro01] M. Castro and B. Liskov. Byzantine fault tolerance can be fast. Dependable Systems and Networks, pages 513–518, 2001.

[Castro02] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems, 20(4):398-461, November 2002.

[Chockler01] G. Chockler, D. Malkhi, and M. K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In Proceedings of the 21st International Conference on Distributed Computing Systems, pages 11–20, April 2001.

[Cristian95] F. Cristian, H. Aghili, R. Strong and D. Dolev. Atomic broadcast: from simple message diffusion to Byzantine agreement. Information and Computation 118(1):158–179, April 1995.

[Dabek01] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide area cooperative storage with CFS. In Proceedings of the 18th ACM Symposium on Operating System Principles, pages 202–215, October 2001.

[Dai-a] W. Dai. Crypto++. http://cryptopp.sourceforge.net/docs/ref/.

[Dai-b] W. Dai. Crypto++. http://www.cryptopp.com/.

[Dwork88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. Journal of the ACM 35(2):288–323, April 1988.

[Ellison99] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. SPKI Certificate Theory, September 1999. RFC2693.

[Feldman87] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. IEEE Symposium on Foundations of Computer Science, pages 427–437. IEEE, 1987.

[Fischer85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. Journal of the ACM, 32(2):374–382. ACM Press, April 1985.

[Frolund03] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: enterprise storage systems on a shoestring. Hot Topics in Operating Systems, pages 133–138. USENIX Association, 2003.

[Fry04] C. P. Fry and M. K. Reiter. Nested objects in a Byzantine quorum-replicated system. In Proceedings of the 2004 IEEE Symposium on Reliable Distributed Systems, pages 79–89, October 2004.

[Ganger03] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. Self-* Storage: brick-based storage with automated administration. Technical Report CMU-CS-03-178. Carnegie Mellon University, August 2003.

[Gasser90] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy, pages 20–30, May 1990.

[Golding95] [9] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J.Wilkes. Idleness is not sloth. Winter USENIX Technical Conference, pages 201–212. USENIX Association, 1995.

[Gong89] L. Gong. Securely replicating authentication services. International Conference on Distributed Computing Systems, pages 85–91. IEEE Computer Society Press, 1989.

[Goodson03] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. Technical report CMU-PDL-03-104. CMU, December 2003.

[Goodson04a] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. International Conference on Dependable Systems and Networks, 2004.

[Goodson04b] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. The safety and liveness properties of a proto- col family for versatile survivable storage infrastructures. Technical report CMU–PDL–03–105. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, March 2004.

[Gray96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. ACM SIGMOD International Conference on Management of Data. Published as SIGMOD Record, 25(2):173-182. ACM, June 1996.

[Gribble00] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. Symposium on Operating Systems Design and Implementation, 2000.

[Herlihy87] M. P. Herlihy and J. D. Tygar. How to make replicated data secure. Advances in Cryptology - CRYPTO, pages 379–391. Springer-Verlag, 1987.

[Herlihy90] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3):463–492, 1990.

[Herlihy91] M. Herlihy. Wait-free synchronization. ACM Transactions on Programming Languages, 13(1):124–149. ACM Press, 1991.

[Herlihy03] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. International Conference on Distributed Computing Systems, pages 522-529. IEEE, 2003.

[Hitz94] D. Hitz, J. Lau and M. Malcolm. File system design for an NFS file server appliance. In Proceedings of the 1994 Winter USENIX Technical Conference, pages 235–246, January 1994.

[Howard88] J. H. Howard et al. Scale and performance in a distributed file system. ACM Transactions on Computer Systems, 6(1):51– 81, February 1988.

[Jayanti98] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. Journal of the ACM, 45(3):451–500. ACM Press, May 1998.

[Jiménez-Peris03] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? ACM Transactions on Database Systems (TODS), 28(3):257-294. ACM, September 2003.

[Katcher97] J. Katcher. PostMark: a new file system benchmark. Technical report TR3022. Network Appliance, October 1997.

[Kihlstrom01] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith. The SecureRing group communication system. ACM Transactions on Information and System Security 4(4), November 2001.

[Kravitz93] D. W. Kravitz. Digital signature algorithm. U.S. Patent 5,231,668, 27 July 1993.

[Kubiatowicz00] J. Kubiatowicz et al. OceanStore: an architecture for globalscale persistent storage. Architectural Support for Programming Languages and Operating Systems, 2000.

[Kung81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. ACM Transactions on Database Systems, 6(2):213-226, June 1981.

[Kursawe02] K. Kursawe. Optimistic Byzantine agreement. Symposium on Reliable Distributed Systems, pages 262-267. IEEE, 2002.

[Lamport78] L. L. Lamport. The implementation of reliable distributed multiprocess systems. Computer Networks, 2:95-114, 1978.

[Lamport82] L. Lamport, R. E. Shostak and M. C. Pease. The Byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3):382–401, July 1982.

[Lamport98] L. Lamport. The part-time parliament. ACM Transactions on Computer Systems, 16(2):133-169. ACM Press, May 1998.

[Lampson92] B. Lampson, M. Abadi, M. Burrows and E. Wobber. Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems 10(4):265–310, November 1992.

[Litwin00] W. Litwin and T. Schwarz. LH*RS: a high-availability scalable distributed data structure using Reed Solomon Codes. ACM SIGMOD International Conference on Management of Data, pages 237-248. ACM, 2000.

[MacCormick04] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. Symposium on Operating Systems Design and Implementation, pages 105-120. USENIX Association, 2004.

[Malkhi97] D. Malkhi and M. Reiter. Byzantine quorum systems. ACM Symposium on Theory of Computing, pages 569–578. ACM, 1997.

[Malkhi98a] D. Malkhi and M. Reiter. Byzantine quorum systems. Distributed Computing 11(4):203–213, 1998.

[Malkhi98b] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems, pages 51–58, October 1998.

[Malkhi00a] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large distributed systems. IEEE Transactions on Knowledge and Data Engineering, 12(2):187–202. IEEE, April 2000.

[Malkhi00b] D. Malkhi, M. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. SIAM Journal of Computing, 29(6):1889-1906. Society for Industrial and Applied Mathematics, April 2000.

[Malkhi01] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the Fleet system. In Proceedings of the 2nd DARPA Information Survivability Conference and Exposition, Vol. II, pages 126– 136, June 2001.

[Martin02] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. International Symposium on Distributed Computing, 2002.

[Morris02] R. Morris. Storage: from atoms to people. Keynote address at Conference on File and Storage Technologies, January 2002.

[Moss81] J. E. B. Moss. Nested transactions: An approach to reliable distributed computing. Ph.D. Thesis, Massachusetts Institute of Technology, May 1981.

[Moss86] J. E. B. Moss. An Introduction to Nested Transactions. COINS TR 86-41, University of Massachusetts, Department of Computer Science, September 1986.

[Mullender85] S. J. Mullender and A. S. Tanenbaum. A distributed file service based on optimistic concurrency control. In Proceedings of the 10th ACM Symposium on Operating System Principles, pages 51–62, December 1985.

[Naor98] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. SIAM Journal on Computing, 27(2):423-447. SIAM, April 1998.

[Narasimhan99] P. Narasimhan, K. P. Kihlstrom, L. E. Moser and P. M. Melliar- Smith. Providing support for survivable CORBA applications with the Immune system. In Proceedings of the 1999 IEEE International Conference on Distributed Computing Systems, pages 507– 516, May 1999.

[Noble94] B. D. Noble and M. Satyanarayanan. An empirical study of a highly available file system. Technical Report CMU–CS– 94–120. Carnegie Mellon University, February 1994.

[Pedersen91] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. Advances in Cryptology - CRYPTO, pages 129–140. Springer- Verlag, 1991.

[Pierce01] E. T. Pierce. Self-adjusting auorum systems for Byzantine fault tolerance. Technical Report TR-01-07, Deparment of Computer Sciences, The University of Texas at Austin. March 2001.

[Pittelli89] F. M. Pittelli and H. Garcia-Molina. Reliable scheduling in a TMR database system. ACM Transactions on Computer Systems 7(1):25–60, February 1989.

[Rabin89] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. Journal of the ACM, 36(2):335–348. ACM, April 1989.

[Reed80] D. P. Reed and L. Svobodova. SWALLOW: a distributed data storage system for a local network. In Proceedings of the International Workshop on Local Networks, August 1980.

[Reed83] D. P. Reed. Implementing atomic actions on decentralized data. ACM Transactions on Computer Systems 1(1):3–23, February 1983.

[Reiter94] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In Proceedings of the 2nd ACM Conference on Computer and Communication Security, pages 68–80, November 1994.

[Reiter95] M. K. Reiter. The Rampart toolkit for building high-integrity services. Theory and Practice in Distributed Systems (Lecture Notes in Computer Science 938), pages 99–110, 1995.

[Rhea03] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. Conference on File and Storage Technologies. USENIX Association, 2003.

[Rivest78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21(2):120–126, Feb. 1978.

[Rivest92] R. L. Rivest. The MD5 message-digest algorithm, RFC-1321. Network Working Group, IETF, April 1992.

[Rosenblum92] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. ACM Transactions on Computer Systems, 10(1):26–52. ACM Press, February 1992.

[Rowstron01] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In Proceedings of the 18th ACM Symposium on Operating Systems Principles, pages 188–201, October 2001.

[Ruemmler03a] C. Ruemmler and J. Wilkes. UNIX disk access patterns. Winter USENIX Technical Conference, pages 405–420, 1993.

[Ruemmler03b] C. Ruemmler and J. Wilkes. A trace-driven analysis of disk working set sizes. HPL–OSR–93–23. Hewlett- Packard Company, April 1993.

[Santry99] D. J. Santry, M. J. Feeley, and N. C. Hutchinson. Elephant: the file system that never forgets. In Proceedings of the 1999 Workshop on Hot Topics in Operating Systems, 1999.

[Schneider90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys 22(4):299– 319, December 1990.

[Shrivastava92] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Spiers and A. Tully. Principal features of the Voltan family of reliable node architectures for distributed systems. IEEE Transactions on Computers 41(5):542–549, May 1992.

[Soules02] C. A. N. Soules, G. R. Goodson, J. D. Strunk and G. R. Ganger. Metadata efficiency in a comprehensive versioning file system. Technical report CMU–CS–02–145, Carnegie Mellon University, 2002.

[Soules03] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. Conference on File and Storage Technologies, pages 43–58. USENIX Association, 2003.

[Steiner88] J. G. Steiner, J. I. Schiller, and C. Neuman. Kerberos: an authentication service for open network systems. Winter USENIX Technical Conference, pages 191–202, 9–12 February 1988.

[Strunk00] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. Symposium on Operating Systems Design and Implementation, pages 165–180. USENIX Association, 2000.

[Thambidurai88] P. Thambidurai and Y.-K. Park. Interactive consistency with multiple failure modes. Symposium on Reliable Distributed Systems, pages 93–100. IEEE, 1988.

[VanRenesse04] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. Symposium on Operating Systems Design and Implementation, pages 91-104. USENIX Association, 2004.

[Wang04] X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. Report 2004/199. Cryptology ePrint Archive, August 2004. http://eprint.iacr.org/.

[Wiener98] M. J. Wiener. Performance Comparison of Public-Key Cryptosystems. CryptoBytes 4(1):1–5, 1998.

[Wool98] A. Wool. Quorum systems in replicated databases: science or fiction. Bull. IEEE Technical Committee on Data Engineering, 21(4):3-11. IEEE, December 1998.

[Wylie00] J. J. Wylie et al. Survivable information storage systems. IEEE Computer, 33(8):61–68. IEEE, August 2000.

[Wylie01] J. J. Wylie, M. Bakkaloglu, V. Pandurangan, M. W. Bigrigg, S. Oguz, K. Tew, C. Williams, G. R. Ganger, and P. K. Khosla. Selecting the right data distribution scheme for a survivable storage system. Technical report CMU-CS-01-120, Carnegie Mellon University, May 2001.

[Wylie04] J. J. Wylie, G. R. Goodson, G. R. Ganger, and M. K. Reiter. A protocol family approach to survivable storage infrastructures. FuDiCo II: S.O.S. (Survivability: Obstacles and Solutions), 2nd Bertinoro Workshop on Future Directions in Distributed Computing, 2004.

[Wylie05] J. J. Wylie A read/write protocol family for versatile storage infrastructures. Carnegie Mellon University Ph.D Dissertation. CMU-PDL-05-108, October 2005.

[Zhou02] L. Zhou, F. B. Schneider, and R. V. Renesse. COCA: A secure distributed online certification authority. ACM Transactions on Computer Systems, 20(4):329-368. ACM, November 2002.